Learning FPGAs

Digital Design for Beginners with Mojo and Lucid HDL

Justin Rajewski

O'REILLY®

Learning FPGAs
by Justin Rajewski

Printed in the United States of America

August 2017: First Edition

Revision History for the First Edition
2017-08-02: First Release

http://oreilly.com/catalog/errata.csp?isbn=9781491965498

978-1-491-96549-8

[LSI]

Contents

Preface

Designing digital circuits used to be something that only big companies could afford to do. It used to require creating *application-specific integrated circuits* (ASICs)—taking weeks or months to produce an actual chip, and requiring piles of cash or wiring together tons of individual chips to perform various logic functions. Then the field-programmable gate array (FPGA) was introduced. FPGAs are programmable logic devices. Unlike an ASIC, the function an FPGA performs is determined at runtime, so an FPGA can be configured to act like just about any digital circuit. However, it wasn't until recently that the cost of FPGAs has dropped to a point where they are now affordable for even hobbyists.

An FPGA allows you to design digital circuits. *Digital circuits* are basically just a bunch of logic gates (and, or, nor, etc.) connected together to perform a specific task. The designs that you create can range from something as simple as a counter that blinks an LED to something as complex as a multicore processor.

This book starts at the very beginning, with setting up your environment and getting an LED to turn on. As you develop your skills, you will learn how to perform more-complicated tasks and eventually design your own basic processor.

The board used in this book is the Mojo (along with the IO Shield). I created the Mojo in 2013 as a simple no-shenanigans FPGA development board for the hobbyist. I ran a Kickstarter to gauge interest and fund the initial round of boards. A lot more excitement than I originally expected resulted and has allowed me to continue working on it. My goal is to build a platform so that anyone who wants to learn about FPGAs—and more generally, digital design—can without having to go to college or having a personal mentor. This book is just another step toward this goal.

Expected Background

This book is going to teach you the basics of digital hardware design. This is not a topic for the complete beginner, and some background information is going to be assumed. You should be familiar with electricity (voltage and current) and basic electrical components (resistors, capacitors, transistors, and LEDs). While not strictly required, some programming background will be helpful, especially if it is with embedded microcontrollers such as an Arduino.

The majority of this book uses Lucid. Lucid is a *hardware description language* (HDL) that was designed to be beginner friendly and simpler to use with FPGAs than the more traditional Verilog and VHDL languages. Lucid is similar to Verilog in many ways, and our tools actually translate it to Verilog as an intermediate step during the build process. However, it removes some of the quirks that plague Verilog and makes many of the easy-to-make mistakes impossible.

Check out the Lucid Quick Reference guide in Chapter 13.

Lucid shares similar syntax with programming languages such as C/C++ and Java. Being familiar with one of these can help. However, it is important to remember that Lucid is a hardware description language and not a programming language.

It is important to have a solid understanding of binary, hexadecimal, and decimal number systems.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

A repo of the book example projects is here:

https://github.com/embmicro/book-examples

A repo of all the built-in example projects in the Mojo IDE is here:

https://github.com/embmicro/components-library/tree/master/base/mojo-v3/Lucid

If you are familiar with Git, you can use it to clone the repos. If you aren't, you can follow those links and click the green "Clone or download" button to open a drop-down menu with a link to download a ZIP of all the files.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/learning_FPGAs*.

CHAPTER 1
Introduction

FPGA stands for *field programmable gate array*. That mouthful is simply trying to tell you that you can program an FPGA over and over (field programmable) and that it is more or less just a large array of logic gates (gate array).

Unless you already know what all that means, it is still not very helpful, so let's break it down a little more. An FPGA belongs to a family of devices known as *programmable logic devices*. These devices allow you to design a digital circuit, and the device will *become* that circuit. This works by configuring small blocks (known as *slices*) to perform logic functions and connecting them together to implement your larger design. A more detailed description of how this all works is covered later. You can reconfigure the FPGA as many times as you want, and each time it will become the new circuit without needing to physically change anything!

In this chapter, we will explain a little more about what FPGAs are and what they are good at. We will also walk you through all the steps to set up the necessary software on Windows or Linux. Unfortunately, Macs are not supported by Xilinx's tools, but you can work through a virtual machine running a compatible operating system. You can do with this with VirtualBox and Ubuntu, both of which are free. With a working environment, we will work through a basic project that will turn on an LED when a button is pressed.

The Mojo

Throughout this book, we will be using the Mojo and its IO Shield. The *Mojo*, shown in Figure 1-1, is an FPGA development board that is relatively inexpensive and requires only a computer and a micro USB cable to use. Unlike many other FPGA development boards, it is fairly minimal, with most of the FPGA's pins broken out onto easy-to-use 0.1″ headers (a total of 84 digital I/O pins).

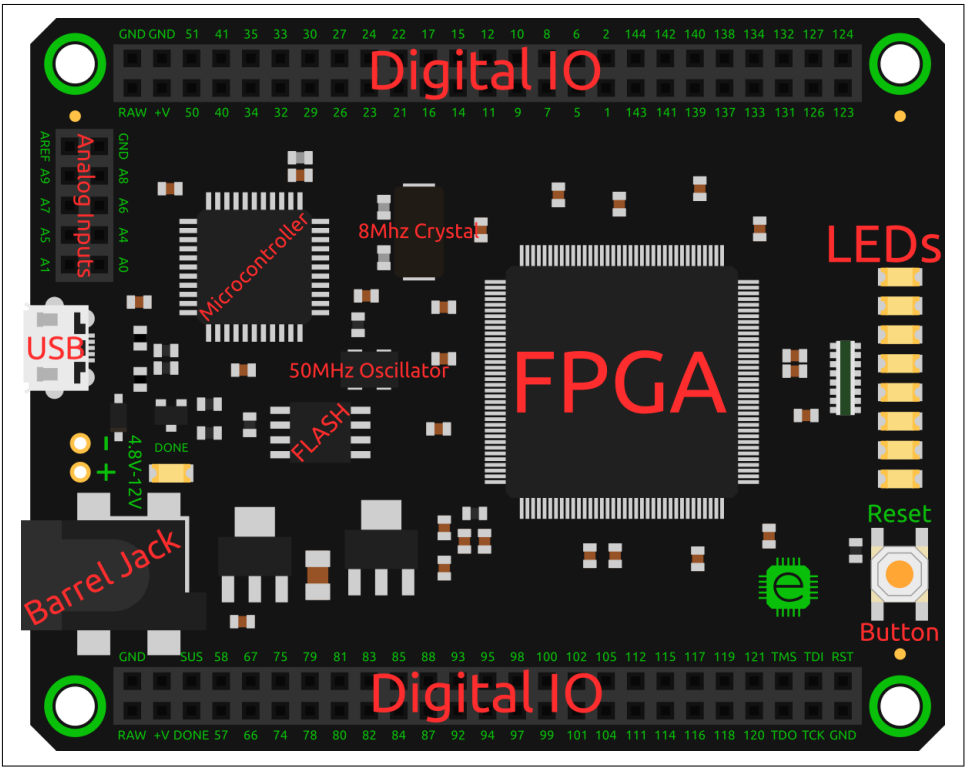

Figure 1-1. The Mojo

The Mojo also has an Arduino-compatible microcontroller whose main function is to program the FPGA over USB. However, once the FPGA is programmed, the microcontroller can be used as an ADC (eight analog inputs are broken out) and a USB-to-serial interface for your FPGA designs. Beyond the other bare necessities such as a 50 MHz oscillator, the Mojo features eight LEDs and a button (commonly used as a reset).

The I/O on the Mojo is all at 3.3 V and is not 5 V tolerant. Power can be supplied via the USB port, the barrel jack, the two holes, or the RAW input on the large headers. The supplied power should be between 4.8 V–12 V, with 5 V being the recommended voltage. You can connect an external power supply and the USB port at the same time, as the USB port is protected by a diode.

You can find more information and order the Mojo from Embedded Micro.

The *IO Shield*, shown in Figure 1-2 is an add-on board that stacks on top of the Mojo. It features 4 7-segment LED digits (like the display on your microwave), 24 LEDs, 24 switches, and 5 buttons. It is a great board for working through example projects. You can find more information and order your own from Embedded Micro.

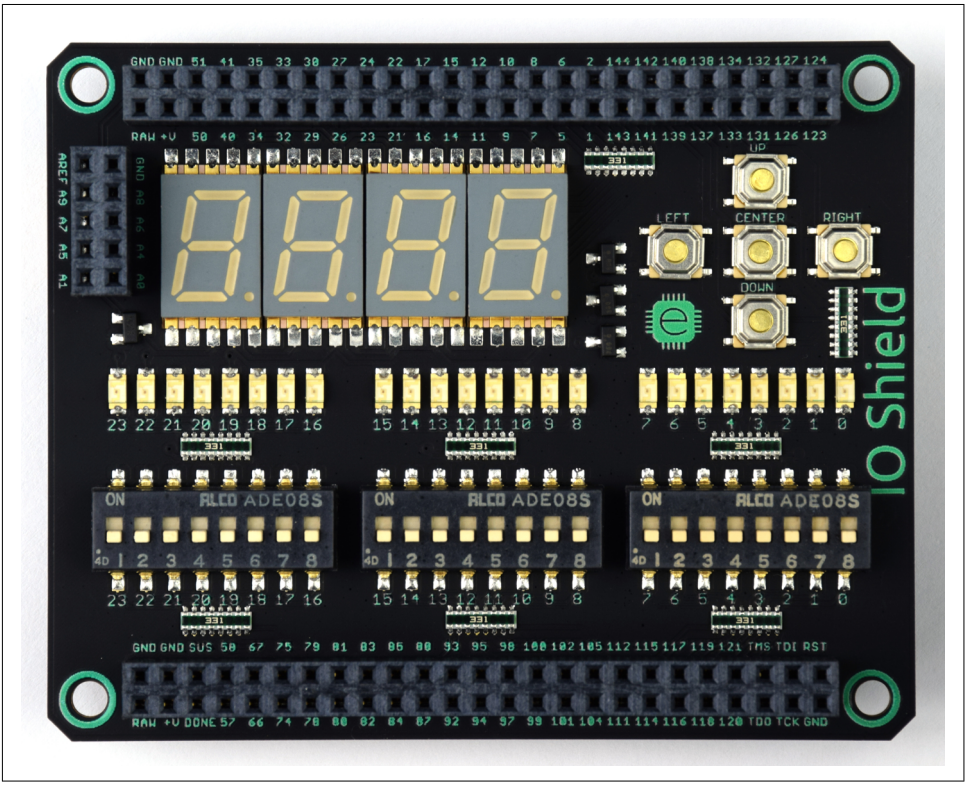

Figure 1-2. IO Shield

FPGAs Versus Microcontrollers: A Comparison

FPGAs and microcontrollers are often compared. This is especially true for people coming from a background of working with boards such as the Arduino or Raspberry Pi. An analogy that I particularly like is to think of a microcontroller as a person. People are flexible. You can teach someone to do almost any task, and his arms and hands are capable of manipulating basically anything. However, people can really focus on only one thing at a time, and unfortunately, we have only two hands. An FPGA, on the other hand, is like an assembly line. You can tailor it specifically for the exact job it needs to do. You can have many stages in the line that are all working at the same time, independent of one another.

FPGAs have been becoming more popular recently and have started appearing in more mainstream news. It used to be that if you wanted to improve the performance of an application, you simply bought or waited for a faster processor. However, in recent years, even though Moore's law continues, processor performance has largely stalled with the extra transistors being used to cram more cores into a single chip.

Because of this, people have started turning to FPGAs to accelerate tasks by creating custom hardware.

When you are working with FPGAs, you are designing hardware, not software. To make this a little clearer, let's look at something that could be done with either an FPGA or a microcontroller: turning on an LED when you press a button. If you were to do this with a microcontroller, such as an Arduino or Raspberry Pi, the code to do this would read the button input, and if it is pressed, turn the LED on, and otherwise turn it off. This would keep happening over and over in a loop. Read button, update LED, read button, update LED, and so on. The problem with this is that the processor is now spending all of its time keeping the LED up to date with the current value of the button.

If you did the same project with an FPGA, you could simply *connect* the LED and the button through the FPGA. Since they are connected, there is no loop, and there is no code; the LED just turns on when the button is pressed. Now, if you wanted to change it up an, say, turn on the LED when the button is not being pressed, with a microcontroller you just change the test condition. However, with an FPGA, you can simply insert a NOT gate between the LED and button.

With the microcontroller, the processor is fully busy with this trivial task. What happens when you want it to also do something else, such as read other inputs or perform calculations? The processor has to juggle the tasks, spending only finite amounts of time on each one. As the complexity of your other code grows, the amount of time the processor has to spend checking the button and updating the LED shrinks, making it take longer and longer for the LED to turn off after you press the button.

FPGAs are not like this. Instead, with an FPGA, since it is all hardware, the circuit you designed to read the button and update the LED will operate completely independently of anything else you throw in your design. You could have another part of the FPGA crunching some serious numbers, but the LED will still be just as responsive as before.

Although you can usually do the same task with a microcontroller or an FPGA, there is usually a clear better choice. Things that require a lot of complex decision-making or sequential operations are commonly better candidates for microcontrollers. Things that can be done in stages or have a regular set pattern are great for FPGAs. This is just a rule of thumb, but the takeaway is that custom hardware and a processor fill different needs, and it's often great to have both (and the Mojo does).

Another powerful feature of FPGAs is that they can read or write every I/O pin every clock cycle. With a microcontroller, you are typically limited to reading registers of 8 or 32 bits that correspond to I/O pins. If you need to sample more pins, you need to do it over multiple cycles, which may not be ideal. This makes it really easy to drive

things that require a lot of I/O, such as LED matrices or a ton of servos. It's also possible to read and write I/O pins at much higher rates. The FPGA on the Mojo is capable of I/O speeds up to 950 MHz; that's 950 million reads or writes a second! Compare this to a typical Arduino's max I/O toggle rate of around 3–4 MHz (with a 16 MHz clock), and this is achievable only if you dedicate all the processing time to toggling the pin.

Table 1-1 summarizes some of the key distinctions.

Table 1-1. FPGA versus microcontrollers

Micro	FPGA
Write software.	Design hardware.
Typically executes one instruction at a time.	All parts of your circuit can operate independently.
Fixed maximum clock speed.	Maximum clock speed is dependent on your design.
A handful of I/O pins that can be accessed in small groups (typically eight) at a time.	Many I/O pins that can all be accessed simultaneously.
Usually store programs in nonvolatile (persistent) memory.	RAM based and needs to be programed after power on (the Mojo does this automatically).
Often very power efficient with advanced sleep modes.	Power usage depends on your design but typically requires more than a microcontroller.
Fixed peripherals that limit the devices you can connect.	Can interface with virtually any digital device.

So what can you do with an FPGA that you can't with a microcontroller? In Chapter 12, we use seven microphones to detect the directions of potentially multiple sound sources. Figure 1-3 shows the layout of the microphones on the board. This is done by calculating the delays between all the microphones for various frequencies and piecing them together to get an image of the sounds around the microphone array. A crucial step is sampling all the microphones at exactly at the same time. With an FPGA, it is fairly trivial to exert precise control over I/O pins. Once the samples are gathered, the FPGA is capable of performing all the calculations incredibly fast. The result is a sensor that constantly listens to its surroundings, generating direction data at over 80 Hz (limited by the sample acquisition time and not the processing time).

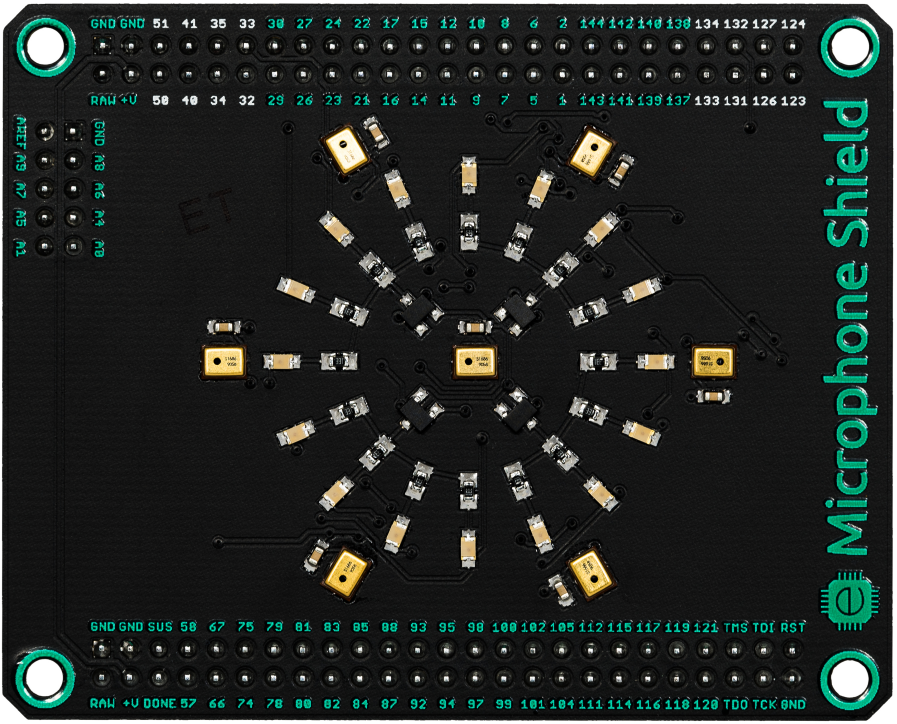

Figure 1-3. Microphone Shield

One of my favorite projects with the Mojo was a hexapod, shown in Figure 1-4. A *hexapod* is a 6-legged robot that usually has 3 servos per leg (18 total) plus 1 or 2 more for the "head." Controlling all these servos can be pretty tricky. Many people have made hexapods without using FPGAs. However, even if all you want to do is make the hexapod walk, you still need to get external servo controllers to control that many servos. With an FPGA, you simply add a servo controller as part of your FPGA design, without the need for any external hardware. With an FPGA, every single I/O pin can control a servo. What made the FPGA in this project really necessary was the addition of a 2-megapixel camera (1,600 x 1,200) that took 15 pictures per second (nearly 29 million pixels per second!). The images from the camera were streamed directly into the FPGA, which performed blob detection (to look for red objects). By having the robot detect red objects, it was able to move its body and "watch" a red ball as someone moved it around. You can check it out in action on YouTube.

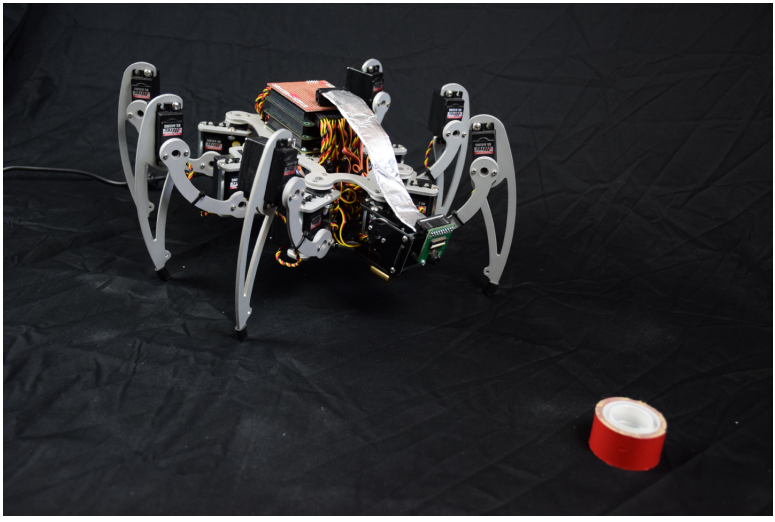

Figure 1-4. Hexapod

Setting Up Your Environment

The first step on your journey to becoming a digital hardware master is to install all the necessary tools. The FPGA on the Mojo is a Spartan 6 XC6SLX9 from Xilinx. Because of this, you need to download and install Xilinx's tools to synthesize (build) your projects into configuration files (often called *bit files*) for the FPGA. Xilinx's tools are a bit clunky, so we will be using the Mojo IDE to do all the work. Although you won't need to use Xilinx's tools directly, the Mojo IDE relies on them to build your designs.

Synthesize

We use the term *synthesize* instead of *compile* and *link*, as you would for code, because the tools for hardware are quite different from software. With software, each file is compiled into instructions that are more or less a direct translation of your code. Each file is then linked with the other compiled files to create your program.

With an FPGA, the tools use your design files as a description of the behavior the circuit needs to have. It then looks at your entire design and synthesizes a circuit that will match its behavior. The tools we are using need to rebuild the entire project if you edit even the smallest thing in a single file. This can be a major headache for bigger FPGAs, when synthesis can take hours or days. Partial resynthesis to speed things up is an active area of development. Luckily for us, most projects on the Mojo synthesize within a few minutes.

The tools require you to be using Windows or Linux. If you're using Windows, Windows 7 or newer is recommended. Although Windows 8 and above isn't officially supported by the tools, it should still work with some minor workarounds. Most versions of Linux should work fine, but CentOS is known to have issues with setting up the Mojo and is not recommended. Ubuntu and its derivatives should work out of the box.

If you have a Mac, you can run Linux in a virtual machine or boot Linux from a USB drive. You can do the same for Windows if you have a license. If going this route, I recommend VirtualBox and Ubuntu. Both are free and known to work.

Installing ISE

ISE is the name of the tool Xilinx provides to build projects for the Spartan 6 family of FPGAs. Xilinx has a newer tool called Vivado, but that doesn't support Spartan 6 devices.

Unfortunately, the installation process for ISE is a bit complicated. You are required to make an account with Xilinx to verify what you are using ISE for and where you live (export restrictions are imposed by the US government). ISE is a fairly large program. The download file is just over 6 GB, and the actual installation requires an additional 16 GB or so of space. After it is installed, you will then need to license it. The FPGA on the Mojo is covered by Xilinx's free WebPACK license, but you still have to get the free license.

Head over to the Xilinx Downloads page. You need to download version 14.7 of ISE. This is the latest and final version. It should already be selected. Scroll down past the Multi-File Download: ISE Design section to ISE Design Suite. In this section, select the full installer for your platform (Linux or Windows). You will likely be prompted to fill out a survey on your reasons for downloading it. Then you will be asked to sign in. Create an account to continue with the download.

If you are on Windows, you need a way to extract the TAR file. Some TAR extractors don't like this archive and fail to extract the entire thing. 7-Zip is known to work and is free. You can download it from the 7-Zip website.

With the files extracted, you need to start the installer. On Windows, double-click the *xsetup* executable. On Linux, you need to run *xsetup* with root privileges:

```
sudo ./xsetup
```

Once in the setup, accept all the license agreements. On the page that asks which edition to install, select ISE WebPACK, as shown in Figure 1-5.

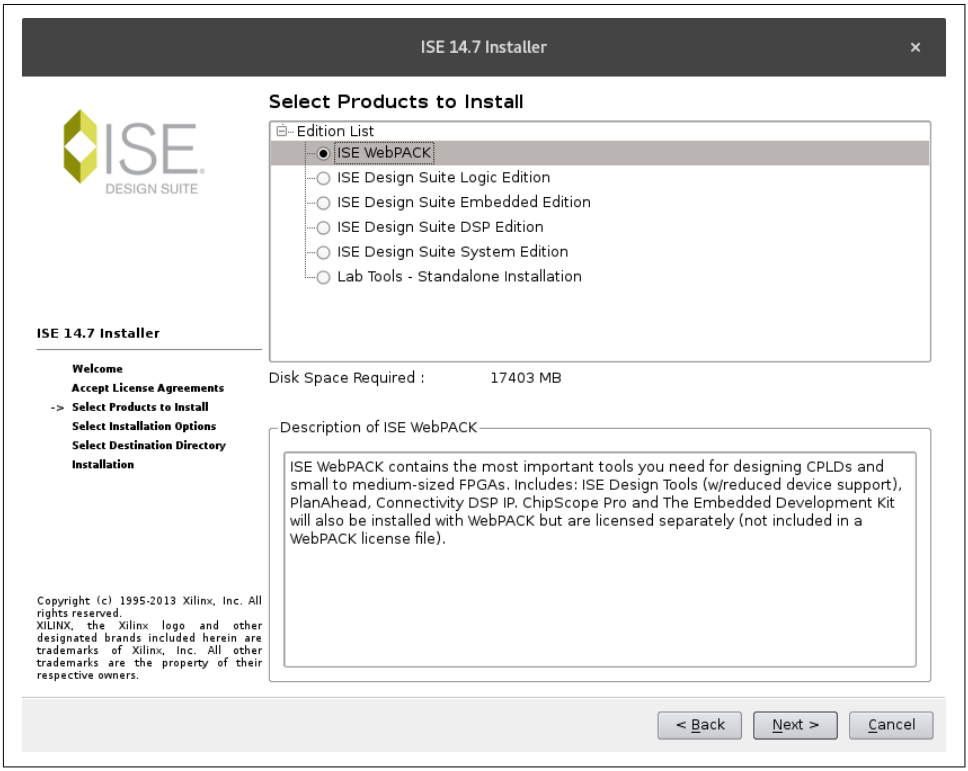

Figure 1-5. ISE edition selection

The next page, shown in Figure 1-6, asks which components you want to install. Make sure that Acquire or Manage a License Key is selected. If you are on Linux, make sure Ensure Linux System Generator Symlinks is also selected, as shown in Figure 1-7. You can uncheck the other options. They won't hurt anything if you install them, but they aren't needed. If you run into trouble with the installer, you can try to uncheck "Use multiple CPU cores for faster installation," as it has been reported to cause problems in rare circumstances.

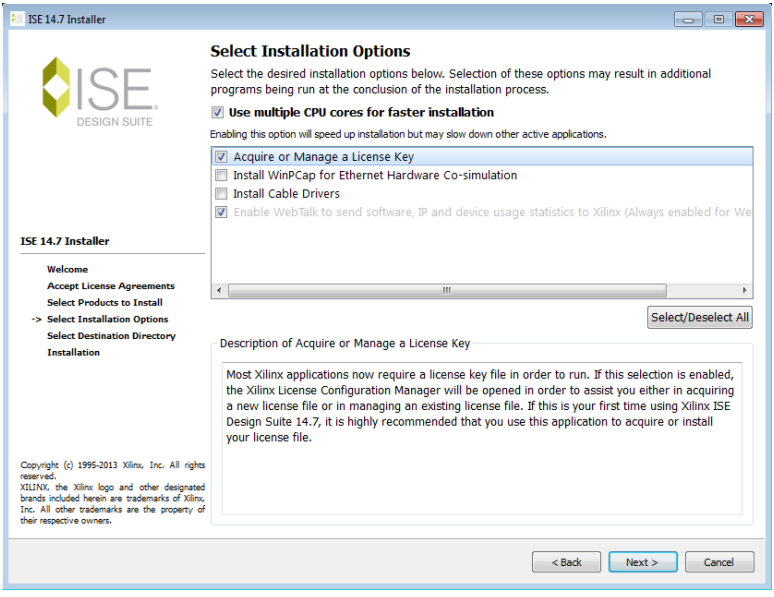

Figure 1-6. ISE installation options for Windows

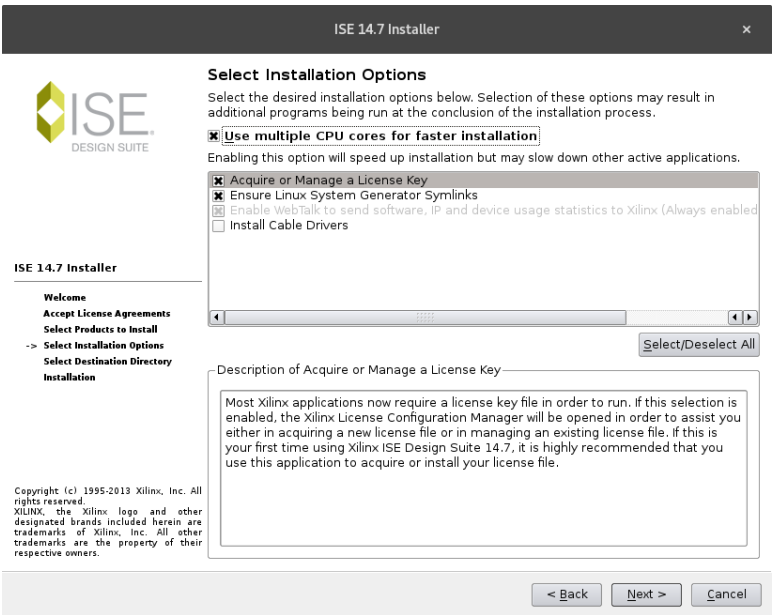

Figure 1-7. ISE installation options for Linux

On the next page, leaving the installation path as the default will make things a little easier later, but feel free to change it as long as you remember where it is.

If you are using Windows 8.1 or Windows 10 on a 64-bit machine and encounter problems, check out Embedded Micro's ISE Installation Tutorial for other OS specific fixes.

Once the installation is complete, a new window to set up a license *should* open. If for whatever reason it doesn't, you can manually open it by launching ISE and going to Help → Obtain a License Key.

On Windows, you can double-click the ISE icon that should now be on your desktop. On Linux, if you installed ISE to the default directory, you can run the following commands to launch it on a 64-bit system:

```
source /opt/Xilinx/14.7/ISE_DS/settings64.sh
/opt/Xilinx/14.7/ISE_DS/ISE/bin/lin64/ise
```

To launch ISE on a 32-bit system, use the following:

```
source /opt/Xilinx/14.7/ISE_DS/settings32.sh
/opt/Xilinx/14.7/ISE_DS/ISE/bin/lin/ise
```

In the Xilinx License Configuration Manager, shown in Figure 1-8, select Get Free Vivado/ISE WebPack License and click Next.

A little dialog box showing your system's information pops up. Click Connect Now to open a web browser to acquire the license. If the button fails to open a browser, you can go to the license page directly.

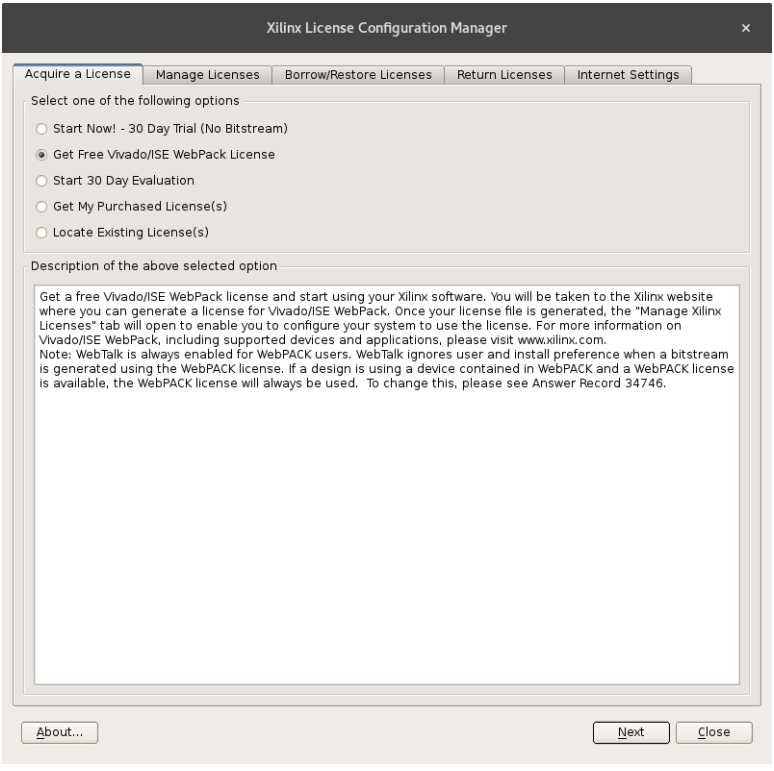

Figure 1-8. Select the WebPack license

You need to log into your Xilinx account again. When you are on the Product Licensing page, shown in Figure 1-9, select ISE WebPACK License and click Generate Node-Locked License to create your license. If you don't see this option, you may have already generated a license before. In this case, head over to the Manage Licenses tab and download your existing license by using the tiny download button in the bottom-left corner.

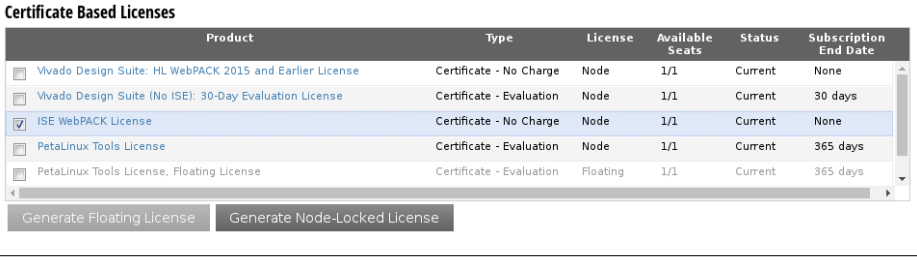

Figure 1-9. Generate the WebPACK license

Click through the little pop-up window until the license is created and you are taken to the Manage Licenses tab. On this page, you should see your newly generated license. To download it, click the tiny blue arrow in the lower-left corner, as shown in Figure 1-10.

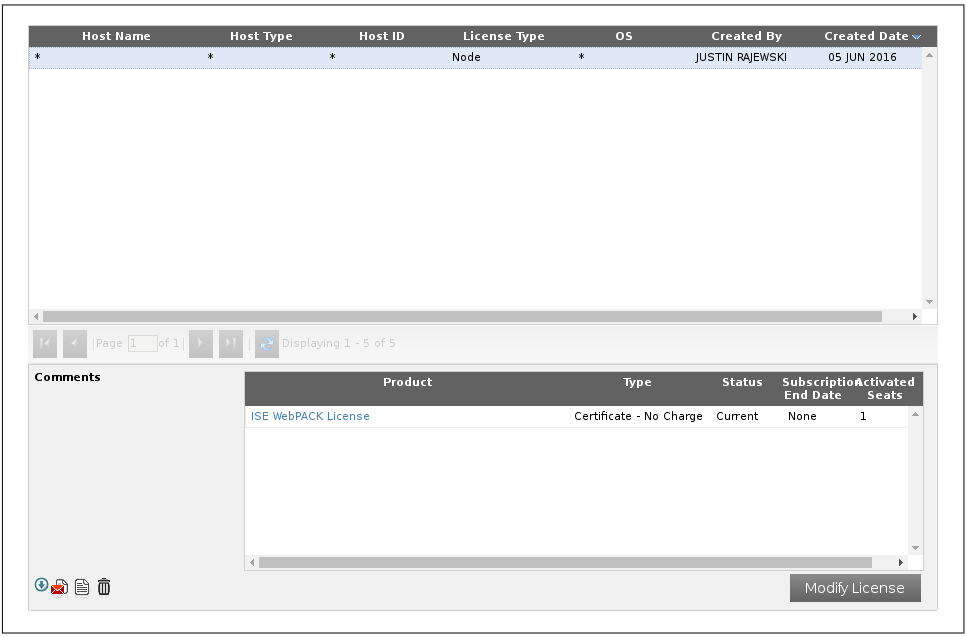

Figure 1-10. Click the little arrow in the bottom-left corner to download your license

A file named *Xilinx.lic* should be downloaded.

Back in the Xilinx License Configuration Manager, navigate to the Manage Licenses tab, if you're not already there, and click Load License. Locate the *Xilinx.lic* file you just downloaded and select it. With the license loaded, the license manager should look like Figure 1-11.

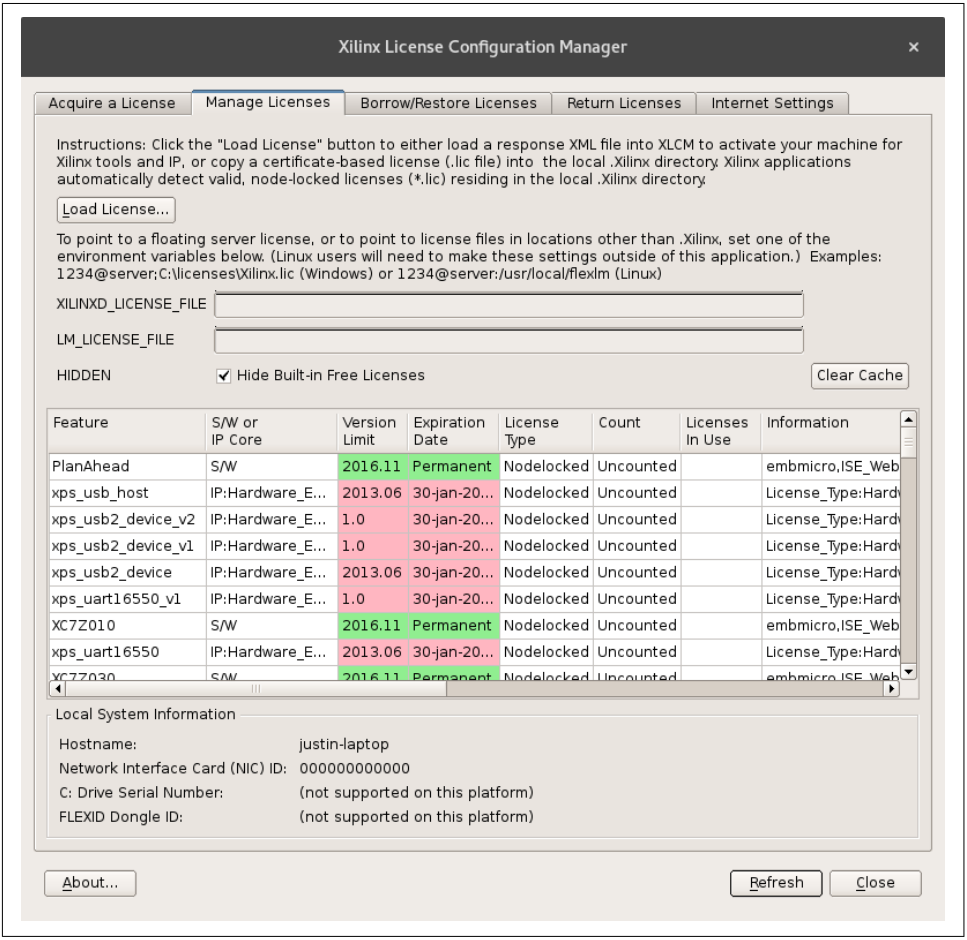

Figure 1-11. License manager with the WebPACK license loaded

Note that it might not look exactly like this, but ISE_WebPACK and PlanAhead should now be green.

ISE should now be ready to build your projects. You can close any ISE-related windows still open.

Installing the Mojo IDE

Installing the Mojo IDE is substantially simpler than setting up ISE. Head over to Embedded Micro and download the latest version. If you are on Windows, the installer is the recommended way to go.

The installer is pretty straightforward. Just follow the instructions until everything is installed. At the end of the installation, it should also install the necessary drivers.

On Linux, you just have the files and you can place them anywhere you want. You also need to have a JRE version 7 or newer to run the IDE. If you are using Ubuntu, you can install it by entering the following command:

```
sudo apt-get install openjdk-7-jre-headless
```

You also need access to serial ports. Again on Ubuntu, you can use the following to add yourself to the `dialout` group to get permission:

```
sudo usermod -a -G dialout `whoami`
```

In the Linux Mojo IDE files, there is a folder named *driver* with a file *99-mojo.rules*. You should copy this file to */etc/udev/rules.d/*. This ensures that you have permission to connect to the Mojo and prevents Ubuntu from thinking it is a modem and locking it up for a while whenever it is plugged in.

You should log out or restart your computer to make sure all these changes take effect.

Mojo IDE Settings

Before you create your first project, you need to tell the Mojo IDE where you installed ISE and what serial port the Mojo connects to.

Launch the Mojo IDE. If you installed it using the Windows installer, you should have a Start Menu entry and/or a desktop icon that you can use to launch it. If you downloaded the Windows files, you can double-click the *mojo-ide.exe* file after extracting the archive. On Linux, you need to run the *mojo-ide* script. This can be done with the following command:

```
./mojo-ide
```

You will be greeted by a welcome dialog box that has a little info about the version of the IDE you downloaded. You can get this welcome dialog back by going to Help → About Mojo IDE.

Go ahead and plug in the Mojo if you haven't already. You need to figure out what serial port the Mojo is connected to. On Windows, you need to open the Device Manager. Launch the Control Panel and navigate to Hardware and Sound. You should then see Device Manager under Devices and Printers.

In the Device Manager, expand Ports (COM & LPT). You should see an entry such as Mojo V3 (COM3), as shown in Figure 1-12. The number following COM may be different. This is the port it is connected to. Remember this.

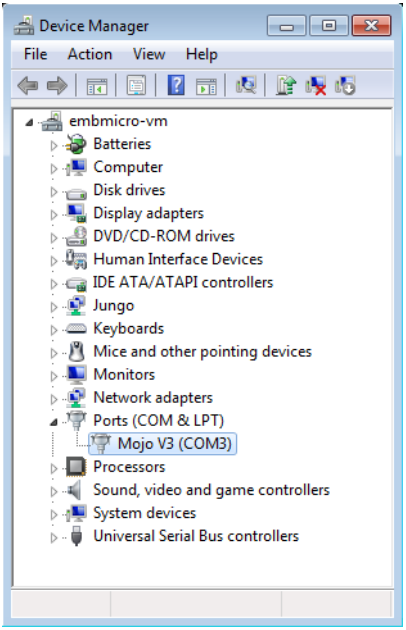

Figure 1-12. The Mojo is connected to COM3. The COM port number may be different for you.

On Linux, the Mojo will most likely connect to */dev/ttyACM0* or */dev/ttyUSB0*. If you have more than one USB-to-serial device, the last number may be nonzero. You can list the ports on your computer with the following command:

```
ls /dev/tty*
```

If you unplug the Mojo and list the ports, and then plug it back in and list them again, the new one is what the Mojo is connected to.

Back in the Mojo IDE, open the Serial Port Selector by going to Settings → Serial Port and select the correct serial port, as shown in Figure 1-13.

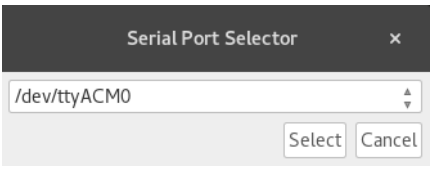

Figure 1-13. Select the serial port you found in the preceding step

The last step is to tell the IDE where to find the Xilinx tools. If you installed them in the default location, *C:\\Xilinx\14.7* on Windows or */opt/Xilinx/14.7* on Linux, then

you can skip this step because the IDE will check these by default. Otherwise, go to Settings → ISE Location and locate the *14.7* directory in your installation.

Everything should now be good to go! It's time to start your first project.

CHAPTER 2
Your First FPGA Project

Now that everything is set up, we are going to create a basic design that will turn on an LED when you press a button. Although this is a fairly trivial example, it presents a lot of new things that will take some time to understand.

Creating a New Project

In the Mojo IDE, choose File → New Project to open the New Project dialog box, shown in Figure 2-1.

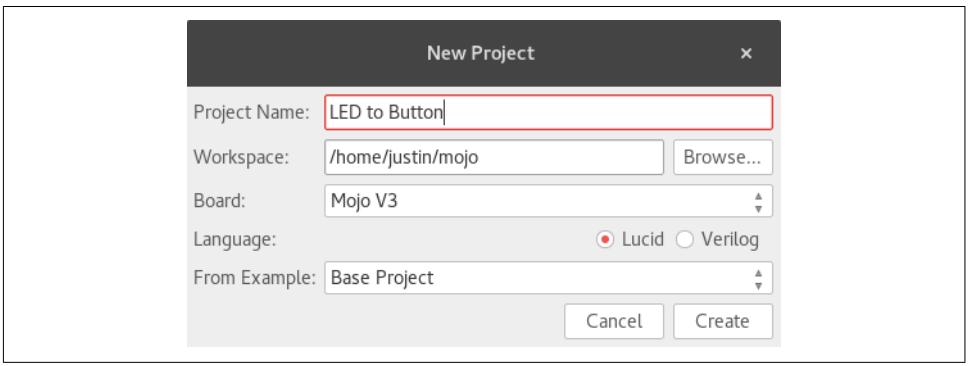

Figure 2-1. Creating a new project

You can make the project name whatever you want, but "LED to Button" fits well. The *Workspace* is the directory where all your projects will be created. In this example, the new project will be in */home/justin/mojo/LED to Button*. Note that if you are using Windows, your path will look different. Make sure Lucid is selected for the language. The From Example option sets what your new project will be based on. There is no fully blank project because there are a lot of connections to the FPGA internal to

the Mojo that you always want to have defined. The Base Project is the most basic template and does nothing except provide a boilerplate design.

Click the Create button to create a new project.

Your project should be created and opened. In the left project tree, expand the project, and then expand Source and double-click *mojo_top.luc*, as shown in Figure 2-2.

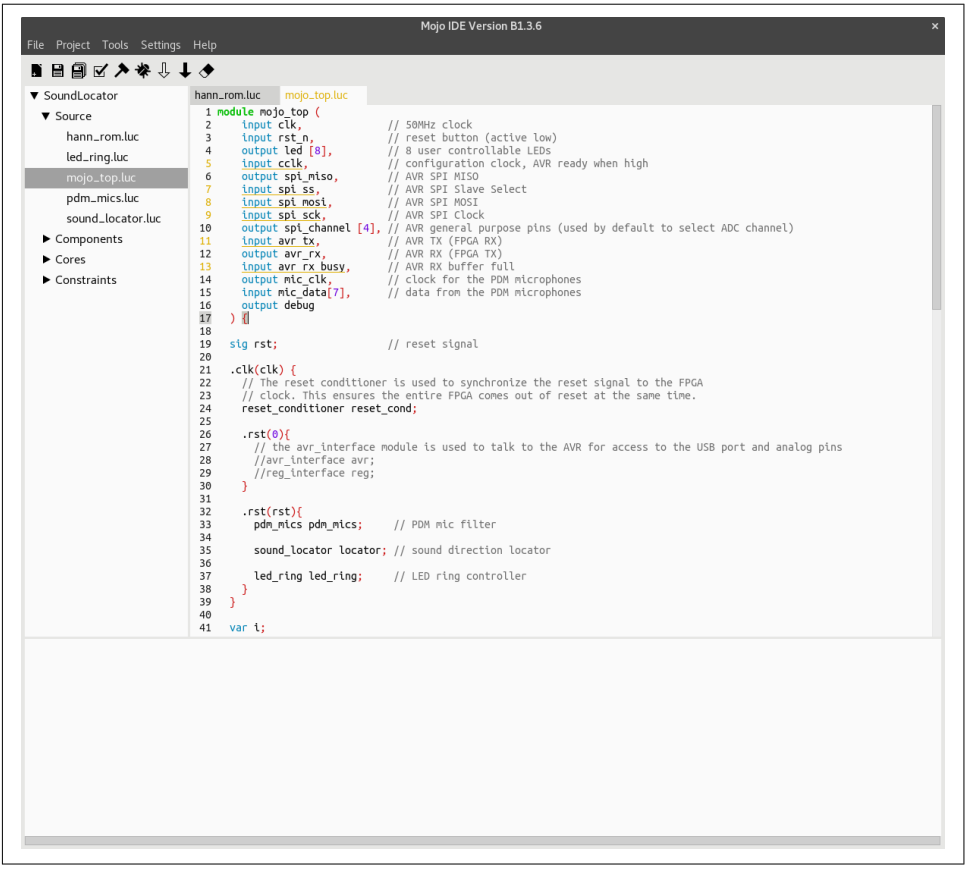

Figure 2-2. New project with mojo_top.luc open

Notice that the name of the open tab is yellow and some of the lines have yellow highlighting. These represent warnings. If you hover your cursor over one, a tooltip will pop up, telling you the actual warning. In this case, these are all warnings about unused inputs. You can also click the little checkbox icon in the toolbar to run a check on your design, and all the warnings and errors will be printed to the console.

Understanding Modules

Hardware designs are broken into modules. This is similar to how programs are broken down into functions. A *module* is a circuit with a set of inputs and outputs. In this example, we have only one module called `mojo_top`. This is the *top-level module* of our project. That means that all the inputs and outputs to the module are actual inputs and outputs of the FPGA instead of internal signals. Later we will instantiate modules inside the top-level module, and all their inputs and outputs will be internal to the FPGA.

Take a look at the port declarations. These are all the connections internal to the Mojo. For this project, we care only about `rst_n` and `led`. Besides `clk` (which is covered later), all the other ports connect to the microcontroller on the Mojo. Note that microcontroller is an AVR (ATmega32U4), and the `avr` prefix on some of the port names refers to this. The connections to the AVR are used to give the FPGA access to the USB port as well as the analog inputs. The `_n` of `rst_n` is there to indicate that it is active low: when the button is pressed, it will have a value of 0 (active), and when the button is just sitting, it will have a value of 1 (inactive):

```
module mojo_top (
    input clk,              // 50MHz clock
    input rst_n,            // reset button (active low)
    output led [8],         // 8 user controllable LEDs
    input cclk,             // configuration clock, AVR ready when high
    output spi_miso,        // AVR SPI MISO
    input spi_ss,           // AVR SPI Slave Select
    input spi_mosi,         // AVR SPI MOSI
    input spi_sck,          // AVR SPI Clock
    output spi_channel [4], // AVR general purpose pins
                            // (used by default to select ADC channel)
    input avr_tx,           // AVR TX (FPGA RX)
    output avr_rx,          // AVR RX (FPGA TX)
    input avr_rx_busy       // AVR RX buffer full
  ) {
```

There are three types of ports: *input*, *output*, and *inout*. By far the most common are inputs and outputs. These are unidirectional and are fairly straightforward to use. Inouts, on the other hand, are bidirectional. Outputs, like inouts, can be tri-stated (output driver is disabled). However, output values can never be read. FPGAs don't have a way to have an internal signal be bidirectional, so inouts can be routed only directly out of the FPGA. Again, this is covered later.

In the FPGA, we call all of these *signals*. It is similar in concept to a variable in code, but they are actual wires carrying signals instead of a value in memory. Signals in a design are a single bit wide unless you specify them to be an array. The output `led` is an example of this. The `[8]` that comes after the name specifies that it is 8 bits wide. Each bit of the array corresponds to an LED on the Mojo.

One of the major benefits to breaking your design into modules is that you can reuse them. What makes this really powerful is the use of parameters. This example doesn't show it, but a module declaration can have a parameter list. These are constants that can be set when the module is instantiated and that allow you to write a generic module that can be customized for each situation.

Representing Values

Before moving on to the next section, let's take a minor detour into how values are represented in Lucid. Values are represented using one or more bits. The number of bits a value has is known as its *width*. There are a few ways to specify a value. Some of them allow you to specify a width, while others use an implied width.

The most basic way to represent a value is with a simple number such as 9. When you do this, the width will be the minimum number of bits needed to represent that value.

Sometimes it's easier to specify a number with a different radix (base) than 10. Lucid supports decimal, binary, and hexadecimal. To specify the radix, prefix the value with `d`, `b`, or `h` for decimal, binary, and hexadecimal, respectively. For example, `hFF` has the decimal value 255, and `b100` has the decimal value 4. If you don't append a radix indicator, decimal is assumed.

It is important to remember that all numbers will be represented as bits in your circuit. When you specify a number this way, the width of the number will be the minimum number of bits required to represent that value when using decimal. For binary, it is simply the number of digits; for hexadecimal, it is four times the number of digits. For example, the value `7` will be 3 bits wide (111), `1010` will be 4 bits wide, and hACB will be 12 bits wide.

You will usually want to specify exactly how many bits a value should be. To do this, you prefix the radix letter with the number of bits. For example, `4d2` will be the value 2, but using 4 bits instead of the minimum 2 (binary value 0010 instead of 10). You must specify the radix when specifying the width to separate the width from the value.

If you specify a width smaller than the minimum number of bits required, the number will drop the most significant bits. When this happens, you will get a warning.

Table 2-1 shows different ways to represent the same values.

Table 2-1. Value representations

Decimal	Hex	Binary
6 or d6 or 3d6	3h6	b110 or 3b110
15 or d15 or 4d15	hF or 4hF	b1111 or 4b1111
6d12	6hC	b001100 or 6b001100 or 6b1100

Although all signals in your design will have a value of 0 or 1, you can assign two other values. One is `x`, which generally means *don't care*. It means you want to assign a value, but you don't care what value gets assigned. Why would you do this instead of just assigning something random? By assigning `x`, you give the tools the flexibility to select whatever value is most convenient. It may turn out that assigning a value of 1 makes your design much simpler to realize in hardware, and since you don't care, the tools can use that.

The `x` value will also show up in simulations, and in that case it means the value is unknown. Either something wasn't initialized properly or some of your *don't care* values are floating around. This is another reason to use `x` when you don't think you care about a value. When you simulate, the unknown values will propagate if used (for example, `x + 2 = x`), so you can ensure that the value really doesn't matter.

The last value is `z`, which means *high-impedance*. Assigning `z` means you are effectively disconnecting that signal. It's important to know that FPGAs can't realize high-impedance signals internally. This means the only time you should use `z` is for outputs of the top module (the module that connects to physical I/O pins). The only real use of `z` is to disable outputs. This is useful for creating an open collector output (instead of 0 and 1, you use 0 and `z`) or for disabling pins that may be driven externally.

Using Always Blocks

Let's skip to the `always` block. We will return to the lines before it shortly.

These blocks are where all the magic happens. They are where you can perform computation and read/write signals. The `always` block gets its name because it is *always* happening. When the tools see an `always` block, they need to generate a digital circuit that will replicate the *behavior* that the block describes.

Both Verilog and VHDL have their versions of the `always` block. In Verilog, they are also called *always blocks*. In VHDL, they are called *processes*.

Take a look at the `always` block in this example:

```
always {
  reset_cond.in = ~rst_n; // input raw inverted reset signal
  rst = reset_cond.out;   // conditioned reset

  led = 8h00;             // turn LEDs off
  spi_miso = bz;          // not using SPI
  spi_channel = bzzzz;    // not using flags
  avr_rx = bz;            // not using serial port
}
```

The `always` block is an abstraction that allows us to create complicated designs without having to worry about exactly how it will be implemented. Inside an `always` block, statements that appear lower in the block have priority over earlier statements. This is kind of similar to programming, where if you write to a variable twice, the second write will be the one that persists, but this is just an abstraction. If a value is written twice in an `always` block, it is as if the first line doesn't even exist. Take a look at the following example:

```
always {
  led = 8h00;             // turn LEDs off
  led = 8hFF;             // turn LEDs on
}
```

So what happens when this is synthesized? If you are thinking about this as code, you may be tempted to think that the LEDs would turn on and off continuously, but that's not what happens. Remember, there is no processor running code; instead, a circuit will be made from this block. When the tools synthesize this, the first line will be ignored completely, and the LEDs will always be on. The `led` output would be hard-wired high. Although this example is trivial, and you would never write to the same signal sequentially like this, with conditional assignments (such as `if` statements) this becomes important.

Back to our design, the `always` block assigns values to six signals. Every output in a module *must* be assigned a value in all circumstances. Because the base project does nothing, and these signals are unused, they are assigned reasonable defaults.

Look at the first two lines in the `always` block:

```
  reset_cond.in = ~rst_n; // input raw inverted reset signal
  rst = reset_cond.out;   // conditioned reset
```

On the first line, we are assigning a value to the input `in` of the module `reset_cond`. Modules, such as `reset_cond`, can be used inside other modules. This is similar to how you can call functions from other functions in code. However, unlike code, where you reuse the exact same instructions, the module will be duplicated in hardware each time you use it. This is why when you use a module it is called *instantiating* that module. You are creating another instance of it. We will get into more detail

about how to instantiate a module in the next section, so don't worry about these lines too much.

We are trying to connect the `rst_n` input, which corresponds to the reset button on the board, to `reset_cond.in`. Note that we actually connect `~rst_n` to the input. The `~` operator inverts the signal (1 becomes 0, and 0 becomes 1). This makes it active high (1 means the button is pressed) instead of active low.

The second line connects the output `out` of `reset_cond` to the signal `rst`. The importance of `reset_cond` will become clear later, but for now just know that it cleans up (synchronizes it to `clk`) the button input, and `rst` will be 1 when the reset button is pressed and 0 when it isn't. For this simple example, it isn't needed, and we could use the `rst_n` input directly.

Take a look at the last four lines now:

```
led = 8h00;             // turn LEDs off
spi_miso = bz;          // not using SPI
spi_channel = bzzzz;    // not using flags
avr_rx = bz;            // not using serial port
```

On the first line, we assign the `led` output to all 0s. Remember from the port declaration that `led` is 8 bits wide. On this line we use `8h00` to make it clear we are setting all 8 bits to 0. We could have also just used the value `0`. This is because `0` would become `1d0`, but since `led` is 8 bits wide, it would be padded with 0s to match the width and would become equivalent to `8b00000000` (or `8h00`).

The next three lines are assigning `z` to the signals. This is because these signals are outputs, so they need a value, but we aren't using them in this example. Since they aren't being used, the safest value is `z`. These signals connect to the microcontroller on the Mojo and are used to get access to the USB port and analog inputs.

When using `x` or `z`, constants won't be padded with 0. For example, if you assign `bx` to a 4-bit signal, it will expand to `4bxxxx` and not `4b000x`. This is true only if the most significant bit is `x` or `z`. If you assign `b0x` to a 4-bit signal, it will expand to `4b000x`.

If you look at the full `always` block, you'll notice that there are no redundant assignments. When the design is synthesized, these values will be hardwired to the signals. The `led` output will be tied low, and the other outputs will be left floating (high impedance).

Connecting the Button

We are going to modify the module to connect the reset button to the first LED so that when you push the button, the LED turns on.

The output `led` is 8 bits wide. Each one of these bits corresponds to a single LED on the board. The least significant bit connects to the topmost LED (rightmost when the LED side of the board is facing you). Arrays in hardware are nothing more than a collection of single-bit signals and are really just a way to conveniently group (and do math on) related bits.

Since `led` is 8 bits wide, we need to assign an 8-bit array to it (each bit needs a value). However, the signal `rst` is a single bit wide. To compensate, we use the *concatenation operator*.

To concatenate multiple arrays into a single larger one, you can use the concatenation operator that takes the form *c{x, y, z}*. Here the arrays (or single-bit values) *x*, *y*, and *z* will be concatenated into a single larger array.

We can modify line 28 to concatenate `7h00`, which is seven zeros, with `rst`, which is a single bit wide, to create an 8-bit array with the 7 most significant bits 0 and the least-significant bit connected to `rst`:

```
    led = c{7h00, rst};     // connect rst to the first LED
```

Building Your Project

Go ahead and click the little hammer icon in the toolbar to build your project. As the project builds, you should see a bunch of text printed in the console. Just wait for it to finish building. It should look like Figure 2-3.

The important line here is `impl_1 finished`. This means your project was built without errors. If you are using Lucid, the Mojo IDE should catch any syntax/design errors. However, as your projects get more advanced, it is possible to create designs that can't be realized in the FPGA, and you will need to check the build output for what went wrong.

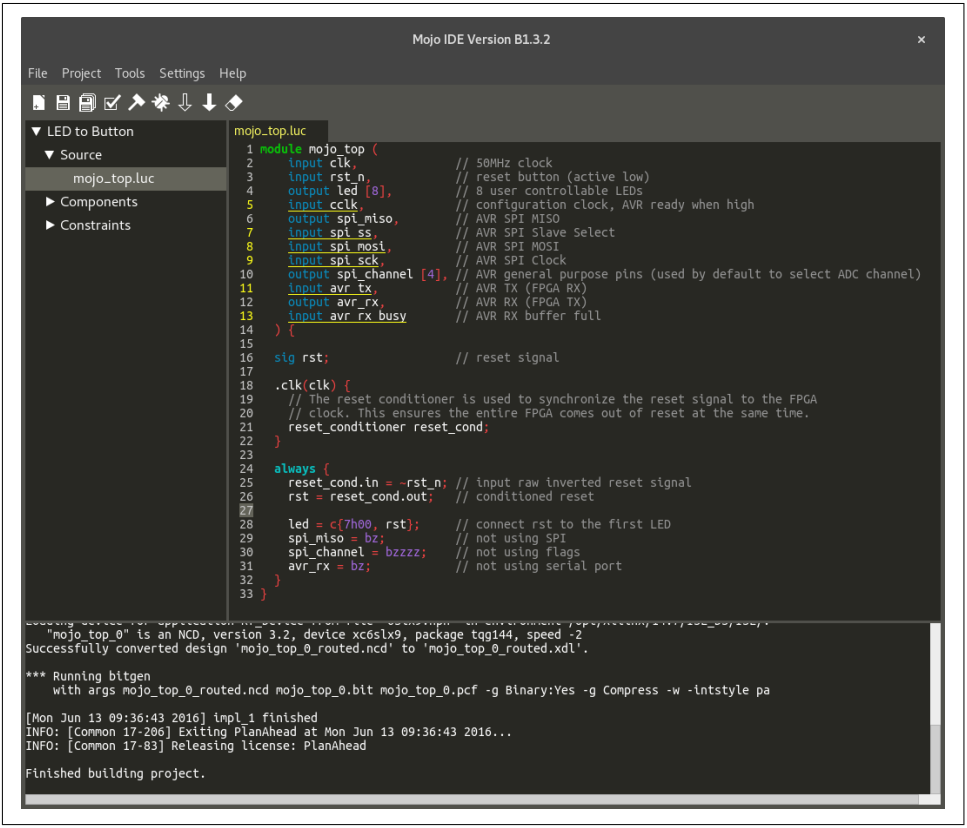

Figure 2-3. Built project with rst connected to led

Loading Your Project

Once your project is built, plug your Mojo into your computer if you haven't already. Also make sure the correct serial port is selected under Settings → Serial Port.

You have two options when programming the Mojo. The first is to load the design directly to the FPGA. This can be done by clicking the hollow arrow icon in the toolbar. FPGAs are RAM-based devices, which means each time they lose power, they also lose their configuration. By loading your design directly to the FPGA, it'll be lost when you disconnect power.

The second option is to program the flash on the Mojo. This will store your design in persistent memory. When the board is powered up, the microcontroller will automatically reconfigure the FPGA. Note that if you program the flash, the microcontroller will program the FPGA at the same time too.

If you have a design loaded in flash and then program the FPGA, when you power cycle the board, the old design in flash will be loaded. This can be helpful if you want to test a design temporarily.

You can also clear the flash by clicking the eraser icon. This will prevent the FPGA from being programmed at power up.

For now, click the hollow arrow to program the FPGA directly, since we will be making some modification soon anyway. After the FPGA is programmed, the IDE should look like Figure 2-4.

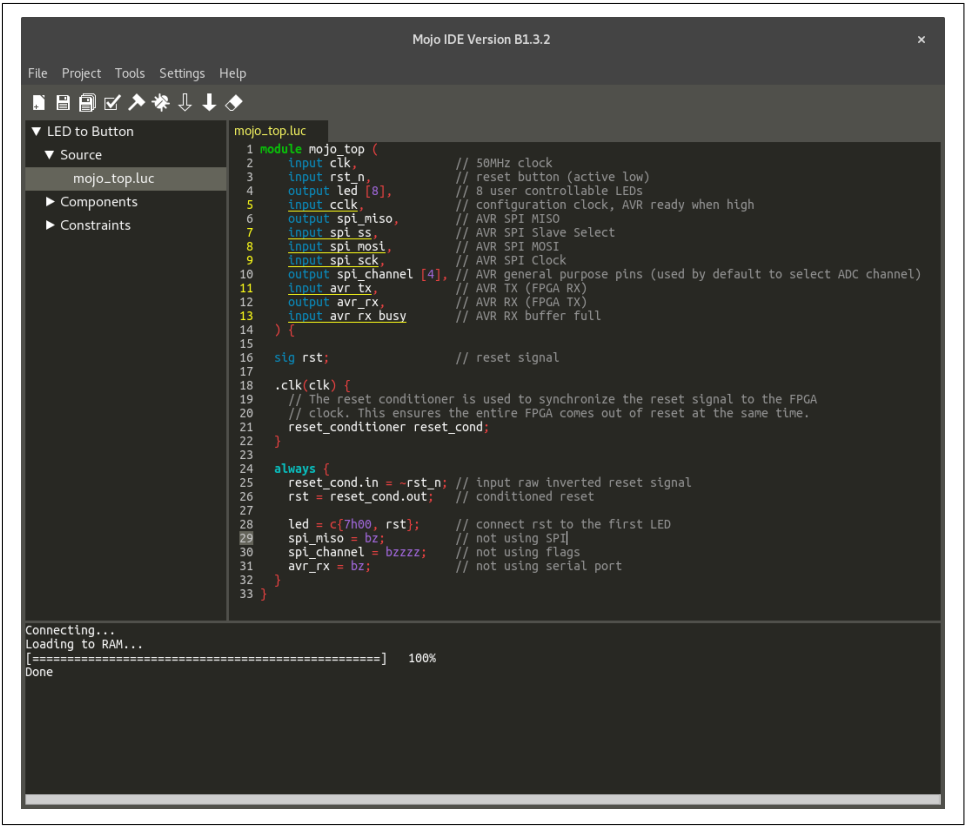

Figure 2-4. Loading the project directly onto the FPGA

If you look at your Mojo, you should now see the *DONE* LED lit. This LED means that the FPGA was successfully configured (Figure 2-5).

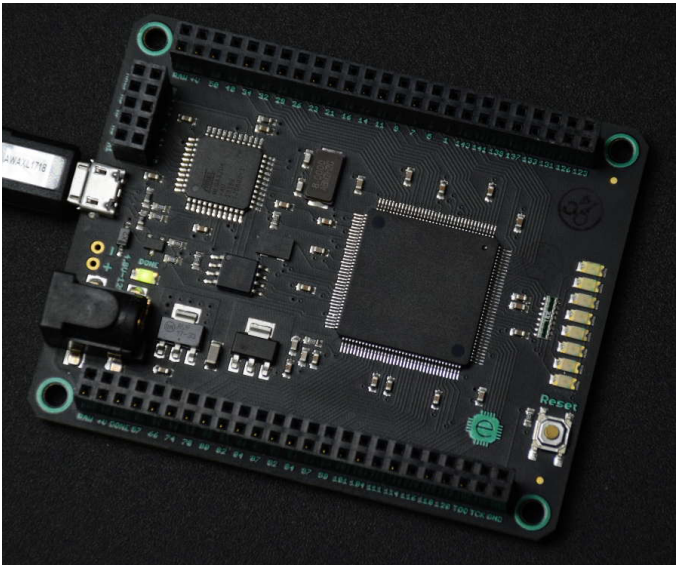

Figure 2-5. Mojo successfully configured

Now push the reset button. The first LED turns on, as shown in Figure 2-6.

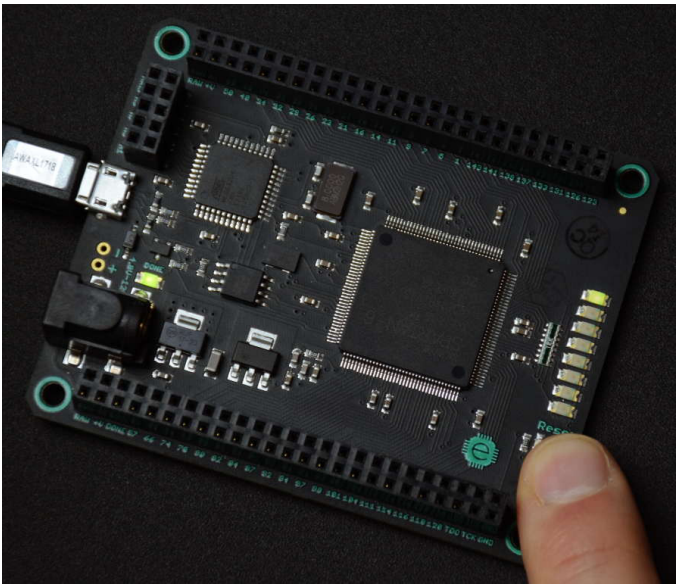

Figure 2-6. The first LED turns on when the reset button is pressed

Some Notes on Hardware

When you press the button, how long does it take for the LED to turn on? If this was a processor instead of an FPGA, the processor would be in a loop reading the button state and turning the LED on or off based on that state. The amount of time between pressing the button and the LED turning on would vary depending on the code the processor was executing and the time it takes to get back to reading the button and turning on the LED. As you add more code to your loop, the variation in delay increases.

However, an FPGA is different. With this design (design, not code), the button input is directly connected to the LED output. You can imagine a physical wire bridging the input to the output inside the FPGA. In reality, it's not a wire but a set of switches (multiplexers) that are set to route the signal directly to the output. Well, this is only partially true, since `reset_conditioner` is there, which does some stuff to clean up the reset signal.

Because the signal doesn't have to wait for a processor to read it, it will travel as fast as possible through the silicon to light the LED. This is almost instant (again, forget about `reset_conditioner`)! The best part is that if you wire the button to the LED then go on to create some crazy designs with the rest of the FPGA; the speed of this operation will not decrease. This is because the circuits will operate independently as they both simply exist. It is this parallelism that gives FPGAs their real power.

Duplication

What if we want all the LEDs to turn on and off with the press of the button instead of just one? Well, we could do it by using the concatenation operator as before, but just use `rst` for each bit:

```
led = c{rst, rst, rst, rst, rst, rst, rst, rst};
```

This is pretty ugly, and there is a much cleaner way to write this. Instead, we can use the *duplication operator* that takes the form `M x{ A }`. Here *M* is the number of times to duplicate *A*. Note that *M* must be a constant. Using this, we can rewrite the line:

```
led = 8x{rst};
```

This line does exactly the same thing as before; it is just a lot cleaner.

Array Indexing

There is another way to write the design so that only one LED turns on. This is by indexing the individual bits in `led`:

```
led[7:1] = 7h0;         // turn these LEDs off
led[0] = rst;           // connect rst to led[0]
```

The first line uses a multibit selector, `[7:1]`. You should read this as "seven down-to one," and it selects the bits 7 through 1. Note that the first value, when using this selector, must be equal to or greater than the second value. Both values must also be constants. These 7 bits are then set to 0.

The second line uses the single-bit selector, `[0]`. This is simply selecting the bit 0, which is then connected to `rst`.

There is another way to select multiple bits. That is by using the *start-width* selector. When using this selector, instead of specifying the start and stop bits (inclusive) as before, you specify the start bit and the number of bits to include above or below it.

The first line from before could be rewritten in any of these three ways, which are depicted in Figure 2-7:

```
led[7:1] = 7b0;  // select bits 7-1
led[7-:7] = 7b0; // select 7 bits starting from 7 and going down
led[1+:7] = 7b0; // select 7 bits starting from 1 and going up
```

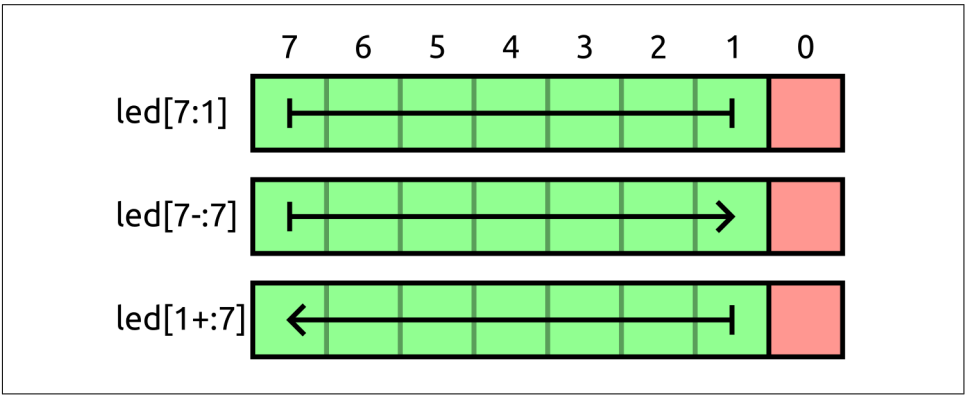

Figure 2-7. Visual of the bit selectors

The main benefit to using the start-width selector is that the start bit can be dynamically specified by a signal. For example, we could use the value of some switch inputs as the start index instead of a constant number. This is because the width of the selection is guaranteed to be fixed (as the width value must be constant) so that it can be realized in hardware.

Always Block Priority

Because of the nature of `always` blocks, you could also write the assignment as follows:

```
led = 8b0;              // turn the LEDs off
led[0] = rst;           // connect rst to led[0]
```

As you may remember, the later assignment to `led` has priority over the earlier one. However, the second assignment assigns only the first bit, so the other seven still retain their values from the first assignment.

Because of the way `always` blocks work, bit 0 of `led` will never have the value 0 and will be directly connected to `rst`. This will create a circuit that is identical to the one before.

Congratulations! You've completed your first FPGA project!

CHAPTER 3

Combinational Logic

Digital circuits can be broken into two main categories: combinational logic and sequential logic (sometimes referred to as *synchronous logic*).

Combinational logic is a digital circuit whose output depends only on the current inputs. The circuit has no memory or feedback loops. It is important to have a strong understanding of combinational logic before proceeding to sequential logic. Even in sequential circuits, you can isolate large sections of combinational logic that perform all the interesting computation.

An example of a combinational logic circuit is the previous button example. The output, if the LED is on, depends only on the current input, if the button is pressed. Another example is a circuit that takes two numbers and adds them together. The output depends only on the two numbers being added. A circuit that would not be considered combinational logic is one that keeps a running total of a series of numbers, as this requires the circuit to remember the current total.

In this chapter, we will run through several combinational logic examples including basic logic gates, various operators, math functions, and more-complex but common circuits. These form the building blocks for all future designs.

IO Shield

The next section requires the IO Shield, shown in Figure 3-1. Alternatively, you could set up a comparable circuit on a breadboard. Go to Embedded Micro for the schematic of the IO Shield. We will be using the DIP switches and LEDs to demonstrate combinational circuits.

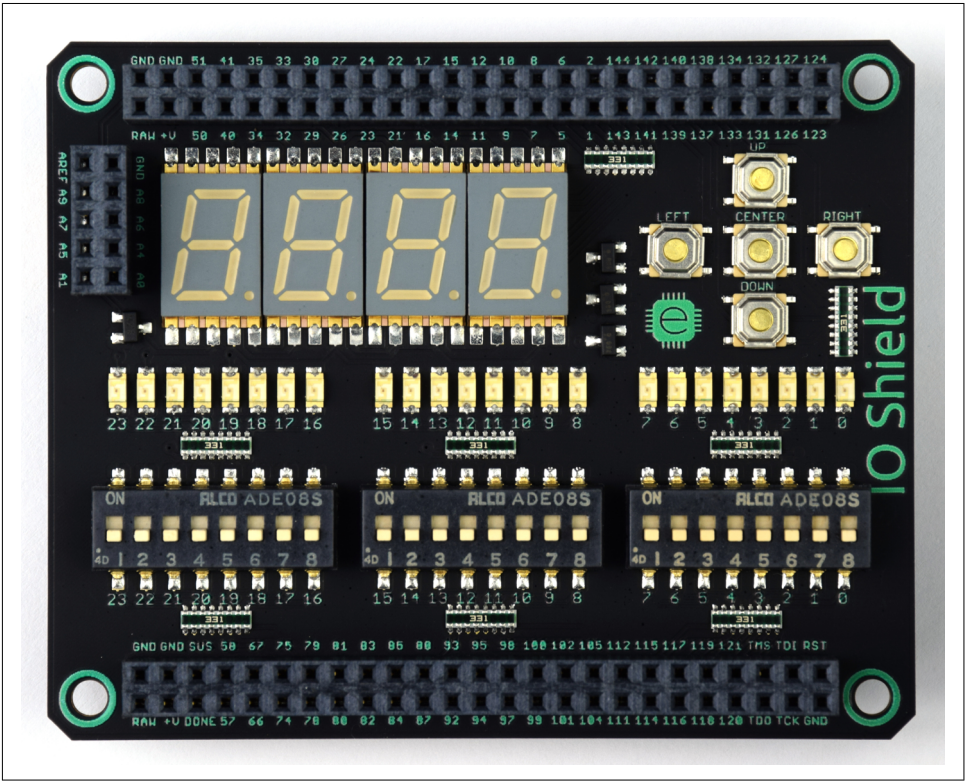

Figure 3-1. IO Shield

We need to make a new project, but this time we are going to base it on the IO Shield Base example project, instead of the Base Project as before. To do this, go to File → New Project, enter whatever name you want ("IO Shield Demo" is a solid choice) and select IO Shield Base from the From Example drop-down menu.

If you open *mojo_top.luc*, you'll notice it is a little different from the version in the bare-bones Base Project we used earlier. We now have some new inputs and outputs declared, and we assign some default values in the `always` block to them. Take a look at the full module, and after we will dive into the details:

```
module mojo_top (
    input clk,              // 50MHz clock
    input rst_n,            // reset button (active low)
    output led [8],         // 8 user controllable LEDs
    input cclk,             // configuration clock, AVR ready when high
    output spi_miso,        // AVR SPI MISO
    input spi_ss,           // AVR SPI Slave Select
    input spi_mosi,         // AVR SPI MOSI
    input spi_sck,          // AVR SPI Clock
    output spi_channel [4], // AVR general purpose pins
```

```
                              // (used by default to select ADC channel)
    input avr_tx,             // AVR TX (FPGA RX)
    output avr_rx,            // AVR RX (FPGA TX)
    input avr_rx_busy,        // AVR RX buffer full
    output io_led [3][8],     // LEDs on IO Shield
    output io_seg [8],        // 7-segment LEDs on IO Shield
    output io_sel [4],        // Digit select on IO Shield
    input io_button [5],      // 5 buttons on IO Shield
    input io_dip [3][8]       // DIP switches on IO Shield
  ) {

  sig rst;                    // reset signal

  .clk(clk) {
    // The reset conditioner is used to synchronize the reset signal to the FPGA
    // clock. This ensures the entire FPGA comes out of reset at the same time.
    reset_conditioner reset_cond;
  }

  always {
    reset_cond.in = ~rst_n; // input raw inverted reset signal
    rst = reset_cond.out;   // conditioned reset

    led = 8h00;             // turn LEDs off
    spi_miso = bz;          // not using SPI
    spi_channel = bzzzz;    // not using flags
    avr_rx = bz;            // not using serial port

    io_led = 3x{{8h00}};    // turn LEDs off
    io_seg = 8hff;          // turn segments off
    io_sel = 4hf;           // select no digits
  }
}
```

If you take a look at where `io_led` is declared, you will notice that it is a two-dimensional array. This corresponds to the three groups of eight LEDs on the IO Shield:

```
    output io_led [3][8],   // LEDs on IO Shield
```

In Lucid, you can have multidimensional arrays that can then be indexed into smaller-dimensional arrays or single bits. For example, `io_led[0]` would select the first 8-bit single-dimensional array.

Because `io_led` is multidimensional, we have to use fancy syntax to assign 0 to all of its bits:

```
    io_led = 3x{{8h00}};    // turn LEDs off
```

The first part of this should seem familiar, as it's the duplication syntax from "Duplication" on page 30. But unlike before, we need to create a 3 x 8 array. If we instead wrote `3x{8h00}`, that would create a 24-bit single-dimensional array of all 0s. We

need to turn the `8h00` from an 8-bit array into a 1 x 8 array. To do this, we use the *array builder operator*, which has the form `{A, B, C}`. Here *A*, *B*, and *C* must all have equal dimensions. For example, `{8h00, 8h37, 8hfa}` would be valid (and would make a 3 x 8 sized array) but `{8h24, 16h2737, 8hfa}` would not be.

The value `{8h00}` then is a 1 x 8 array. By using the duplication operator, we duplicate the outermost dimension, making a 3 x 8 array. We could have also written `{8h00, 8h00, 8h00}`, but the duplication syntax is a little cleaner.

As you may have noticed, the IO Shield has three groups of eight LEDs. The first dimension of `io_led` selects the group, and the second dimension selects the individual LED.

The 2D array `io_dip` is organized the sameway as `io_led`.

Logic Functions

The first example we will work through is to connect two of the DIP switches to an LED through a logic gate. A *logic gate* is a circuit that takes numerous logical inputs (true/false) and has a single output. In digital circuits, 1 is considered to be true, and 0 is considered to be false. The basic logic gates are NOT, AND, OR, and XOR, which are illustrated in Figure 3-2.

NOT takes a single input, and outputs true only when the input is false. In other words, it negates the input.

AND takes two or more inputs, and outputs true only when all the inputs are true. If an AND gate has two inputs, you can think of its output as being true only if the first input *and* the second input are true.

OR takes two or more inputs, and outputs true only if one or more of the inputs are true. If an OR gate has two inputs, you can think of its output as being true if either the first input *or* the second input is true.

XOR takes two or more inputs, and outputs true only when an odd number of inputs are true. This is a little weird for more than two inputs, but if you look at the two-input case, it will output true when either input is true, but not when both are true. XOR is short for e*x*clusive *or*.

There are also variations on AND, OR, and XOR that invert the output. They are known as NAND, NOR, and XNOR, respectively. You can think of them as the base versions followed by a NOT. Their symbols are the same as their parent symbols but with a bubble at the output (like the NOT symbol).

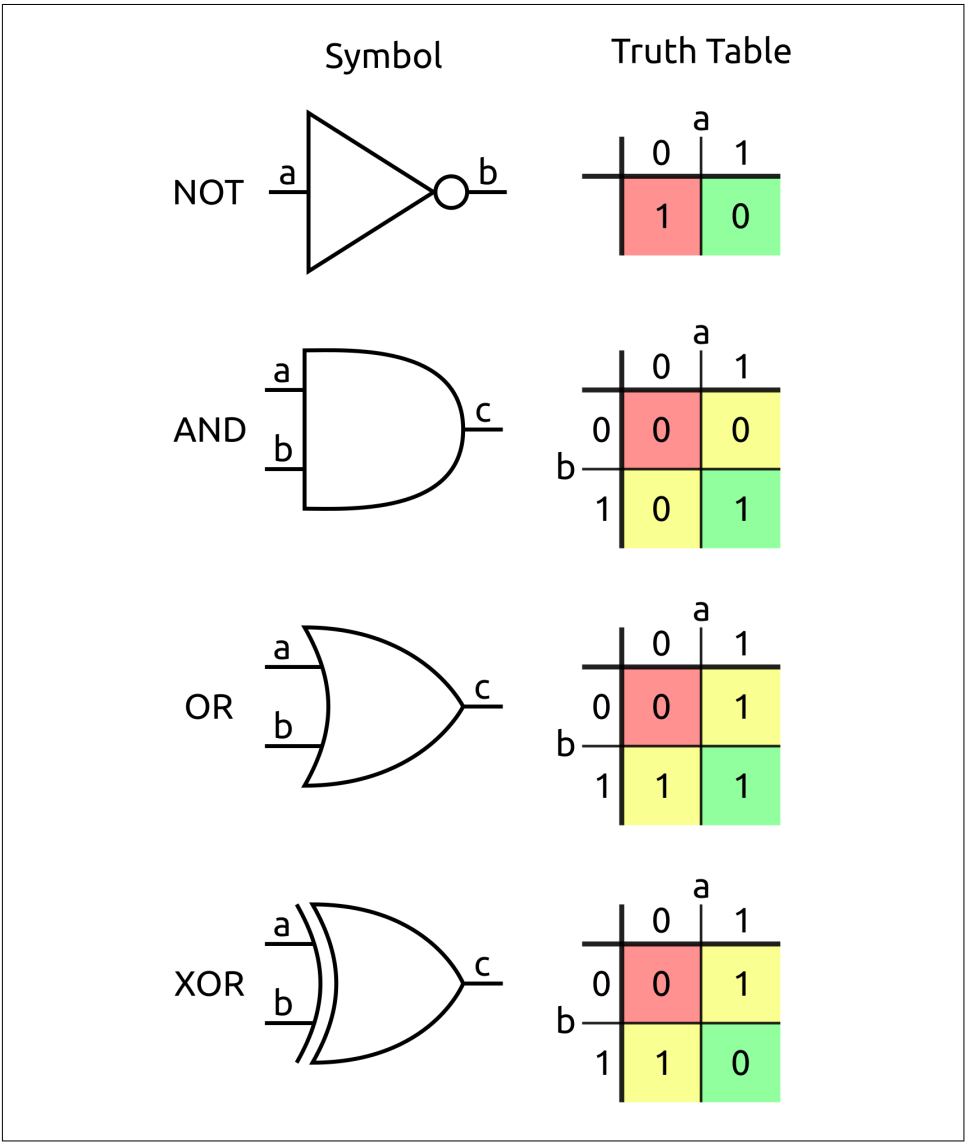

Figure 3-2. Schematic symbols and truth tables for NOT, AND, OR, and XOR

We can add a single line to the end of the `always` block to turn on the first LED when switch 0 and 1 are both on. Note that we are going to consider the order of the switches from right to left, not left to right as printed on the switches themselves. The rightmost switch in a group is switch 0, and the leftmost is switch 7, as shown in Figure 3-3.

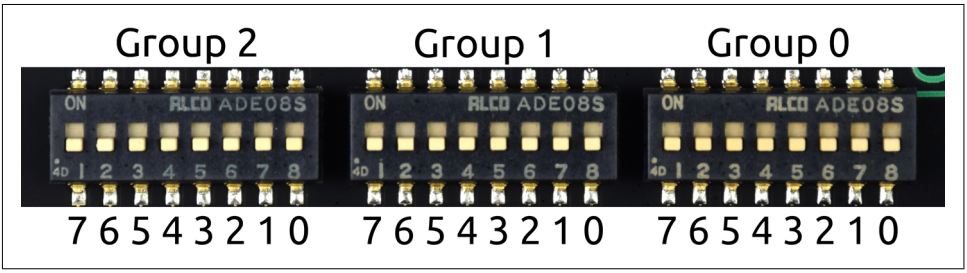

Figure 3-3. IO Shield switch numbering

```
always {
  reset_cond.in = ~rst_n; // input raw inverted reset signal
  rst = reset_cond.out;   // conditioned reset

  led = 8h00;             // turn LEDs off
  spi_miso = bz;          // not using SPI
  spi_channel = bzzzz;    // not using flags
  avr_rx = bz;            // not using serial port

  io_led = 3x{{8h00}};    // turn LEDs off
  io_seg = 8hff;          // turn segments off
  io_sel = 4hf;           // select no digits

  io_led[0][0] = io_dip[0][0] & io_dip[0][1]; // new line!
}
```

Looking at the new line, we need to index the first LED. To do this, we first select the first group, group 0, and the LED is the first in the group, so also index 0. The bit selection is then `[0][0]`. We index the first and second DIP switches in the same manner. Finally, we use `&`, the bitwise AND operator, to AND the bits together.

Notice that the line where all of `io_led` is assigned to 0 is still the same. Trying to assign all the bits but the first would be overly complicated when the new line will simply override the value for the first bit. It is pretty common to have a line early on in the `always` block that assigns a default value.

Build and load the project onto your Mojo. Try playing with the rightmost two switches. If either switch is off, the LED will be off. Only when both are on, the LED should be on.

Go back and replace the `&` operator with the operators for OR and XOR: `|` and `^`. Play with the switches, and make sure you understand how each works.

Practice

Add two more lines to the `always` block so that the first LED of the first group lights up as before, the first LED of the second group lights up when either (OR) of the first two switches in the second group are on, and the first LED of the last group lights up only when exactly one (XOR) switch of the first two switches in the last group are on.

Bitwise Operators

So far, all the operators we have talked about are *bit-wise operators*: they can be used on two equally sized arrays, and they will perform their operation on each pair of bits individually.

For example, if we want to light up an LED in the first group only when the corresponding switch in the first and second groups are both on, we could do the following:

```
io_led[0] = io_dip[0] & io_dip[1];
```

Remember that `io_led[0]` is actually an array of 8 bits. The same is true for `io_dip[0]` and `io_dip[1]`. This single line will create eight AND gates—one for each pair of bits.

You can use the OR and XOR operators in the same way.

We can also chain multiple operators together. For example, if we want the LEDs in the first group to turn on only when the corresponding switches in all three groups are on, we could do the following:

```
io_led[0] = io_dip[0] & io_dip[1] & io_dip[2];
```

Bitwise operators are evaluated from left to right, so in this case `io_dip[0]` will be ANDed with `io_dip[1]`, and the result of that will be ANDed with `io_dip[2]`. In the case of all AND operators, the order doesn't matter. However, if you start mixing operators, the order can matter. You can use parentheses to force the order you want.

You can negate any of these operators (1 becomes 0, 0 becomes 1) by simply using the ~ operator. This will flip each bit in a single array. If we wanted to NAND the first two groups of switches and output the result on the first group of LEDs, we could use the following:

```
io_led[0] = ~(io_dip[0] & io_dip[1]);
```

De Morgan's Laws

De Morgan's laws are the pair of Boolean logic relations that allow you to transform between AND and OR operators. The relations are *~(A & B) == ~A | ~B* and *~(A | B) == ~A & ~B*. Given this, we could have also written the previous line as follows:

```
io_led[0] = ~io_dip[0] | ~io_dip[1];
```

I would read the original line as "true when `io_dip[0]` and `io_dip[1]` aren't both on." I would read the second version as "true when either `io_dip[0]` or `io_dip[1]` are off." However, these are the exact same expressions. You can prove it to yourself with a truth table.

Practice

Use two bitwise operators so that the LEDs in the first group light up when the corresponding switch in the third group is on or when both switches in the first two groups are on.

Reduction Operators

The *reduction operators* are similar to the bitwise operators except they operate on a single array and output a single bit. In other words, they reduce an array to a single bit. You can think of these operators as a single logic gate with as many inputs as the size of the array.

For example, if we wanted to turn the first LED on when all the switches in the first group are on, we could use the following line:

```
io_led[0][0] = &io_dip[0];
```

This line is equivalent to ANDing each bit individually as follows:

```
io_led[0][0] = io_dip[0][0] & io_dip[0][1] & io_dip[0][2] & io_dip[0][3]
               & io_dip[0][4] & io_dip[0][5] & io_dip[0][6] & io_dip[0][7];
```

It should be obvious that the reduction operator is much cleaner.

The reduction operators are not limited to single-dimensional arrays. We could use the OR reduction operator to turn on the first LED when any switch is on with the following:

```
io_led[0][0] = |io_dip;
```

The AND reduction operator is great for checking whether a value is its max value (all 1s). The OR reduction operator will basically tell you if the array isn't zero, and

the XOR reduction operator (^, like the bitwise version) will tell you if there is an odd number of 1s in the array.

Practice

Using your newly acquired bitwise and reduction operator skills, make the first LED light up when any switch in the third group is on, or all the switches in the second group are on and the first group has an odd number of switches on.

Math

We will now look at addition, subtraction, and multiplication by performing operations using the DIP switches as our number inputs and the LEDs as the output. This means everything will be in binary, so make sure you are familiar with it. Check out Embedded Micro if you need some background.

Let's start with some addition. We will add the binary values from the first and second groups of DIP switches.

Because the result of many of our operations will be more than 8 bits, we can create a signal, `result`, to hold the value. You can think of signals (`sig`) as wires. They don't take up any space in your design, as they simply define the connection from something to something else.

We set `result` to be the output of our addition. The addition of two 8-bit numbers results in a 9-bit result. This is because an 8-bit number has the range 0–255, so adding two of them together will have the range 0–510, which requires 9 bits to represent.

We then take `result` and connect it to the LEDs so that we can see all 24 bits:

```
sig result [24];          // result of our operations

always {
  reset_cond.in = ~rst_n; // input raw inverted reset signal
  rst = reset_cond.out;   // conditioned reset

  led = 8h00;             // turn LEDs off
  spi_miso = bz;          // not using SPI
  spi_channel = bzzzz;    // not using flags
  avr_rx = bz;            // not using serial port

  io_seg = 8hff;          // turn segments off
  io_sel = 4hf;           // select no digits

  result = io_dip[1] + io_dip[0]; // add the switch values

  // connect result to LEDs
```

```
    io_led = {result[23-:8], result[15-:8], result[7-:8]};
}
```

Build and load your project to your Mojo. Congratulations! You've just built a basic calculator. To make sure everything is working properly, let's calculate 2 + 2. 2 has the binary representation of 0000 0010. Set both groups of switches to 2. The result on the LEDs should be 4, or 0000 0100.

Now change the `+` to a `-` to subtract the first switch group's value from the second group's value:

```
result = io_dip[1] - io_dip[0]; // subtract the switch values
```

What happens when you set the second switch to 0 and the first to 1 so that you're performing 0 – 1? All 24 LEDs turn on! You can interpret this answer in two ways. The first is that the result simply underflowed and restarted back at the maximum value. The second way is to assume the number is a two's complement representation of –1. Both are valid interpretations, and what you use will depend on the scenario.

Two's Complement

Two's complement is a way to interpret binary numbers so that negative numbers are allowed. It also has some convenient properties such as unsigned addition and subtraction circuits work for signed addition, and subtraction without modification. If the *most significant bit* (MSB) of a two's complement number is 1, it is a negative number, and to get the positive value, you simply invert the bits and add 1. Inverting the bits and adding 1 also works to negate a positive number. For example, the value 1111 would be considered –1. If we invert the bits, 0000, and then add 1, 0001, we get 1. Going in the reverse direction, we get 1110 + 1 = 1111, or –1.

The range for an unsigned n-bit binary value is 0 to $2^n - 1$. An n-bit signed (two's complement) binary value has the range -2^{n-1} to $2^{n-1} - 1$. For example, an 8-bit unsigned number could go from 0–255, while the signed version would go from –128 to 127. Note that two's complement numbers can represent one more negative number than positive, as 0 takes up the last slot.

Even though when the number is negative all 24 LEDs light up, you need only 9 bits to represent any 8 bit by 8 bit addition or subtraction. This is because when the value would be negative, dropping the leading 1s doesn't change its value.

Finally, change the `-` to a `*` to try multiplication:

```
result = io_dip[1] * io_dip[0]; // multiply the switch values
```

You should notice that multiplying two numbers can generate a much bigger result than the operands. The result will generally be twice the size of the inputs—so in our case, up to 16 bits wide.

Multiplication is much more expensive to perform than a simple addition or subtraction. This is because a multiplication is essentially many conditional additions. Multiplying two 8-bit numbers will result in a series of eight additions.

Also note that we didn't talk about division. This is because although there is a division operator, you generally shouldn't use it unless it's only with constants so the tools can replace it at synthesis time. Division is much more complicated than even multiplication, and there are some trade-offs you need to decide for your design. If you don't need real division, you can approximate it by multiplying with fractions.

To approximate division, you can first multiply by any number and then shift the value to the right. Shifting to the right effectively divides by a power of two (and truncates). For example, if you want to divide a number by 3, you could multiply by 86 and then shift right 8 bits. This will effectively multiply your value by 86/256 = 0.33594, which is pretty close to 1/3. If you need higher precision, you can use more bits. For example, you could instead multiply by 21,846 and shift 16 bits to the right. This is effectively multiplying by 0.333389, but 16-bit multiplication is significantly more expensive than 8-bit:

```
x_divided = (x * 16d86) >> 8; // divide x by ~3
```

When selecting a value to use as the numerator, you should use the smallest value that is still larger than the fraction you are trying to approximate. For example, 85/256 is about 0.33203, which is less than 1/3, but 86/256 is about 0.33594, which is just over 1/3. Because the bit shifting will truncate the value, using a slight overestimate will generally perform better. The expression `(8d3 * 16d85) >> 8` is equal to 0, but `(8d3 * 16d86) >> 8` is 1, as it should be.

Notice that the numerator width is set to 16 bits. In this example, `x` and `x_divided` are 8 bits wide, so we want the result of the multiplication to be 16 bits, as we will be shifting off the 8 LSBs. If we used `8d86`, the width of `(x * 8d86)` would be 8 bits wide, and shifting right would result in 0 for any value of `x`.

This is because the width of the multiplication result is dependent on the two numbers being multiplied and the width of the signal being assigned. The width is set to the widest of these three values. If we use `8d86`, the width of all three of these is 8 bits, so the multiplication result is 8 bits wide. By changing to `16d86`, we force the full multiplication result to be used.

Common Subcircuits

There are a handful of common subcircuits you'll see used in designs. All of these are made up of logic gates (as all digital circuits are), but they are often abstracted into their own symbols for clarity in schematics. Here we will cover some of the most common ones, including multiplexers, decoders, encoders, and arbiters.

Multiplexers

Multiplexers are circuits that can select one of many values based on the *selection* input. In Figure 3-4, the multiplexer will select one of the six inputs, a through f, and output it on g. The `sel` input in this case needs to be one-hot.

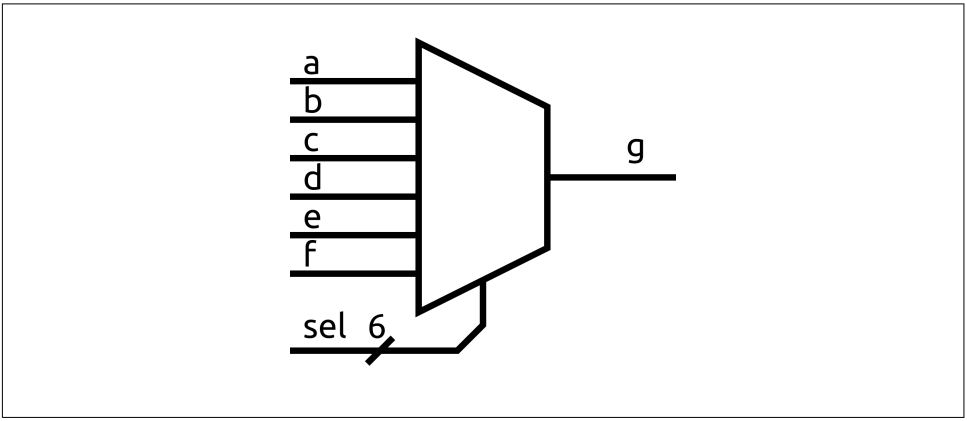

Figure 3-4. Schematic symbol for multiplexers

One-Hot

A *one-hot signal* has exactly one bit that is 1. In the multiplexer, the `sel` input will always be made up of one 1 and five 0s. The position of the 1 corresponds to the input to select.

A variation of one-hot, called *one-or-none*, allows for all bits to be 0.

One-cold is the same idea as one-hot, except there is always exactly one 0.

Although there are many ways to realize a multiplexer, the schematic in Figure 3-5 is one potential way to make one.

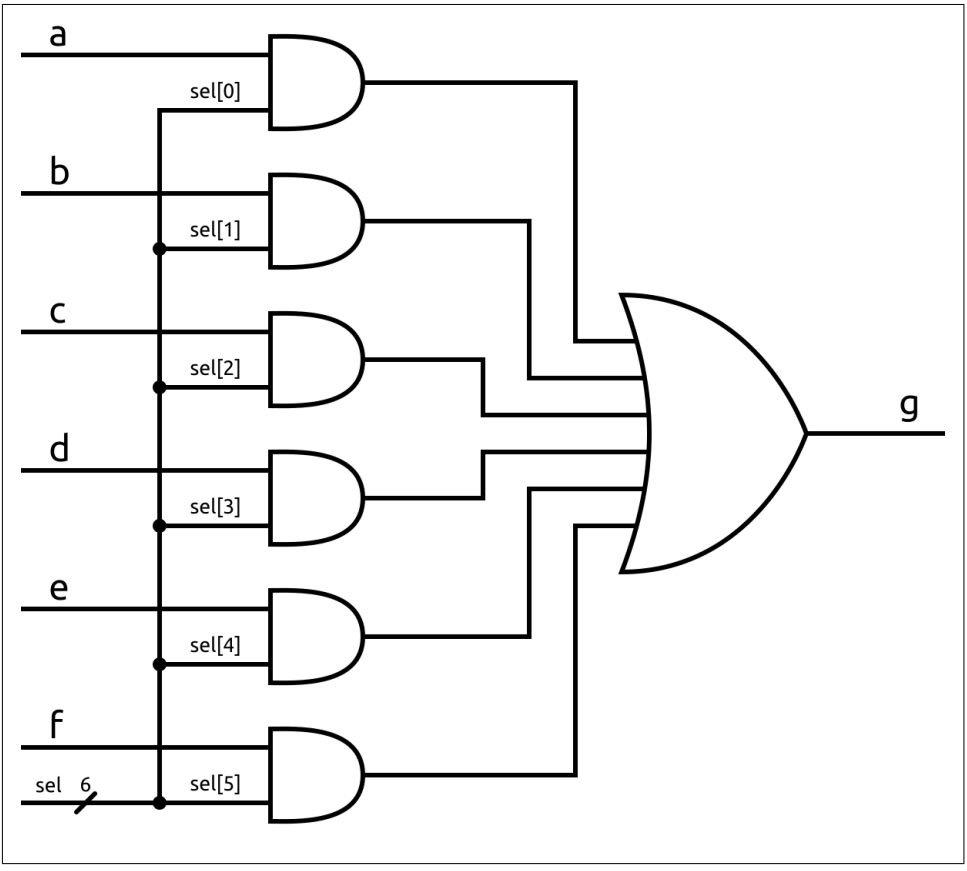

Figure 3-5. Multiplexer circuit

When one of the inputs of an AND gate is 1, its output will be the same as the other input (1 AND 1 is 1, 1 AND 0 is 0). If one of the inputs is 0, the output is always 0. Therefore, the only AND gate that can be outputting 1 is the one with a corresponding `sel` bit set to 1. Even then it outputs 1 only if the corresponding input is 1. By ORing all the AND gate outputs together, we can reduce all the signals to 1 bit. All the inputs to the OR gate will be 0 unless the selected input is 1; then only that input will be 1, but that is enough to make the OR gate output 1.

In your designs, you'll rarely have a module dedicated to being a multiplexer or even think about instantiating them explicitly. This is because `if` and `case` statements (covered later) usually become multiplexers.

Decoders

The multiplexer circuit example requires the select input to be one-hot. Some circuits are simpler when an input is one-hot, but what if we want to use a binary value as the select input? This is where decoders come in. A *decoder* converts a binary (encoded) value into a one-hot representation.

Decoders don't have a special symbol like multiplexers do. Instead, they are commonly drawn in a box with a label specifying that it is a decoder, as shown in Figure 3-6. Notice that the output of a decoder will have a width of 2 to the power of the width of the input.

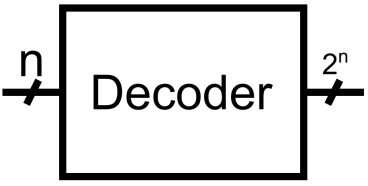

Figure 3-6. Decoder symbol

Figure 3-7 shows one way to make a decoder. It basically amounts to a bunch of equality checks. If the input is 0, the first output is 1. If the input is 1, the second output is 1, and so on. As you may have guessed, a decoder for many bits can get pretty big. An 8-bit decoder would have 256 outputs!

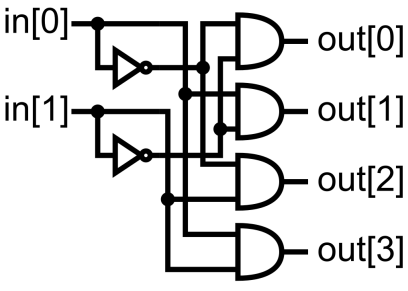

Figure 3-7. Decoder circuit

Encoders

Encoders are the opposite of decoders. They convert a one-hot value into a binary representation.

Just like decoders, encoders don't have a special symbol and are drawn using a labeled box, as shown in Figure 3-8. However, they take a power of 2 width input and have an output width of that power.

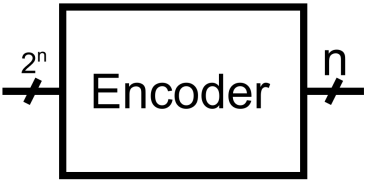

Figure 3-8. Encoder symbol

The schematic for an encoder is a bit simpler than the decoder's, as you can see in Figure 3-9 This implementation uses OR gates to set all the bits high in the output for that specific value. Note that bit 0 of the input isn't even used. This is because when this bit is 1, the output should be 0.

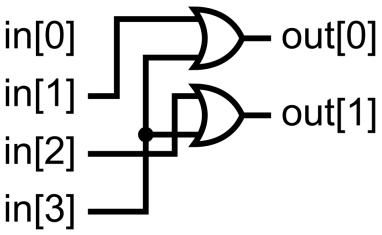

Figure 3-9. Encoder circuit

Arbiters

Sometimes you will have multiple parts of your design fighting over a resource. In this case, it can be helpful to use an arbiter. An *arbiter* takes a set of inputs and then outputs a one-or-none-hot value. The bit that is 1 in the output will be the most significant bit in the input that is 1. For example, if you have a 4-bit arbiter with an input

`4b0111`, the output would be `4b0100`. You can think of the input as four things asking for permission, with a 1 meaning permission requested. Each input has priority over the bits lower than it (the MSB has the highest priority). The arbiter then grants permission to the highest-priority input requesting permission. It will grant permission only to a single input, but if none are requesting it, it will output 0. This is where the one-or-none-hot name comes from: one or none of the bits will be 1.

This may not be the best scheme for sharing resources for every application because a greedy high-priority input could starve out the other inputs. However, unlike something like round-robin, where each input takes turns, no state needs to be saved, and the circuit is fairly simple.

The structure of an arbiter is known as a *ripple-carry chain*. The reason for this will become obvious when we look at how to make one.

At each input bit, we need to know only if there is a 1 above us and if the input is a 1. If there isn't a 1 above us and the input is a 1, we can output a 1. If the input is 0 or there already is a 1, we need to output 0. However, how do we know if there is a 1 above us? If we look at the MSB (`in[3]` in Figure 3-11), there obviously isn't a 1 above us because there are no bits higher than the MSB. We can then tell the next bit that there aren't any 1s above it if the MSB isn't a 1.

At any bit, we can tell the bit lower than it that there isn't a 1 if there isn't a 1 above the current bit and the current bit isn't a 1. We can build a circuit, as shown in Figure 3-10, that takes these two inputs, the current input bit and the bit telling us whether permission has already been granted, and produces two outputs, indicating whether this bit gets permission and whether permission has already been granted (including to the current bit). The input and output bits that signal whether a 1 has been seen are called *carry* bits.

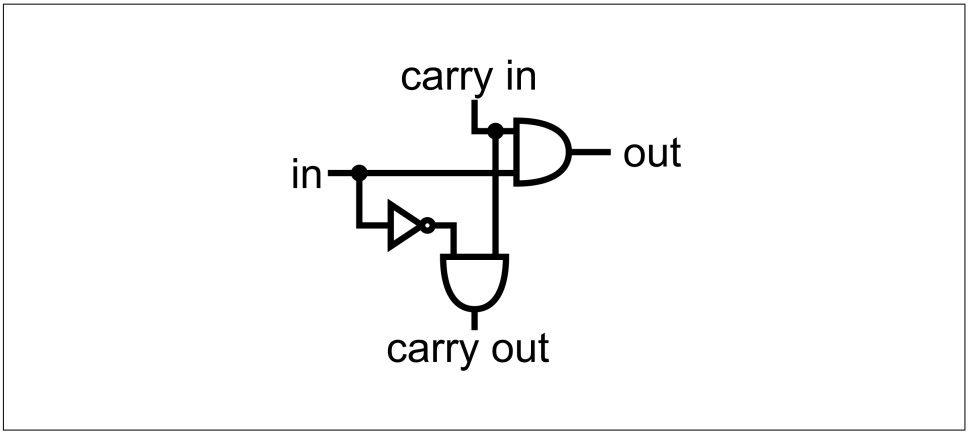

Figure 3-10. Single bit in arbiter circuit

We can then take this circuit and duplicate it for each bit, as shown in Figure 3-11. We chain the *carry* bits together. Note that since there aren't any bits above `in[3]` to feed into the carry, we need to initialize it. Here we feed in a constant 1 to signal that no 1s have been seen before.

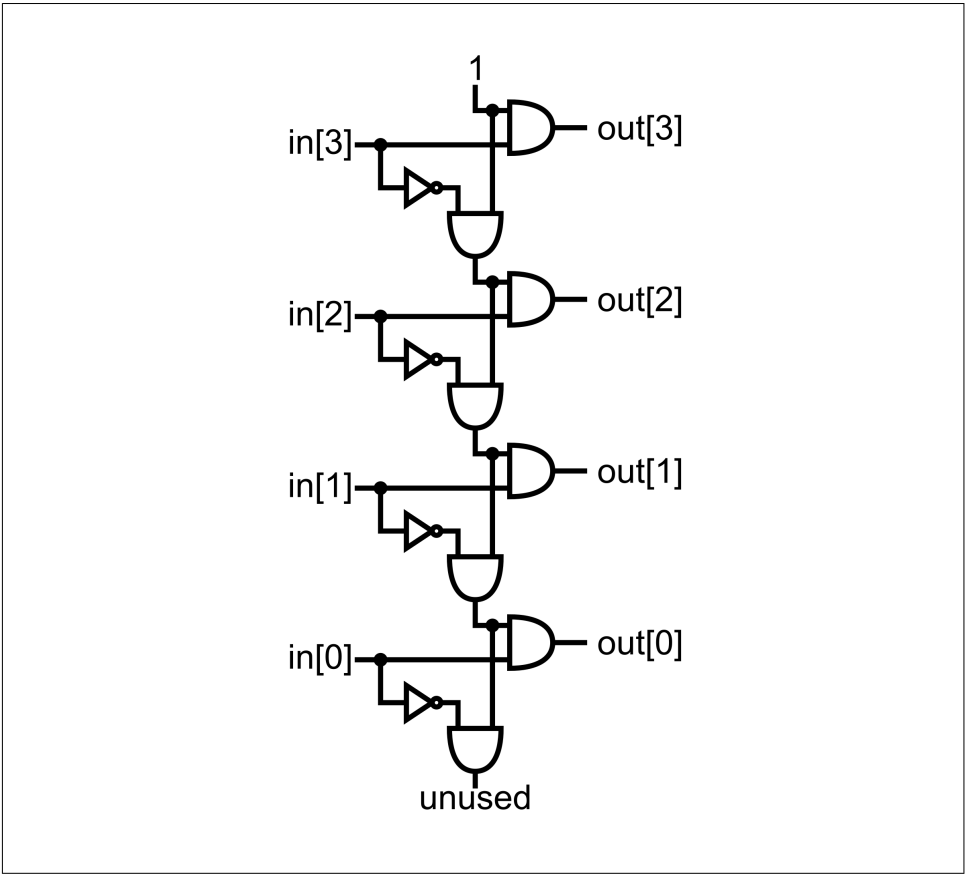

Figure 3-11. 4-bit arbiter

The first and last bits in the chain can be simplified, as shown in Figure 3-12 because we don't need to output a carry at the end, and the value of the initial carry is known.

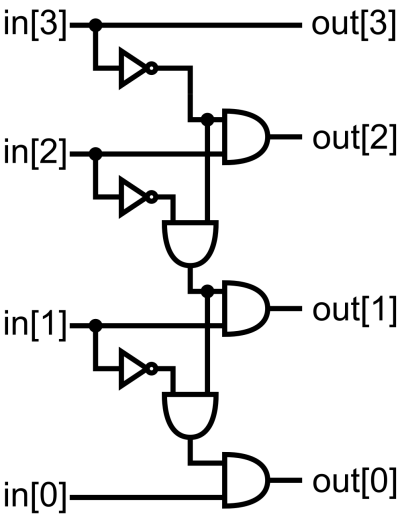

Figure 3-12. Simplified 4-bit arbiter

Notice that for the least significant bit's value to be determined, every other bit needs to be determined. This makes longer chains take more time to produce a result.

Decoder, Encoder, and Arbiter Example

In this example project, we are going to instantiate a decoder, encoder, and arbiter and hook them up to the switches and LEDs of the IO Shield. The Mojo IDE has components for all of these circuits built into something known as the *Components Library*. We will use these premade components, but first we need to create a new project.

Create a new project (I called mine DecEncArb), and base it off the IO Shield Base example project.

Components Library

The *Components Library* has a ton of built-in modules that you may find helpful in your projects. We are going to take a look at how to add these components to our project.

To access the Components Library, go to Project → Add Components. We need three of the components for this project: the *Decoder*, *Encoder*, and *Arbiter*. All of these can be found under the Miscellaneous category, as shown in Figure 3-13.

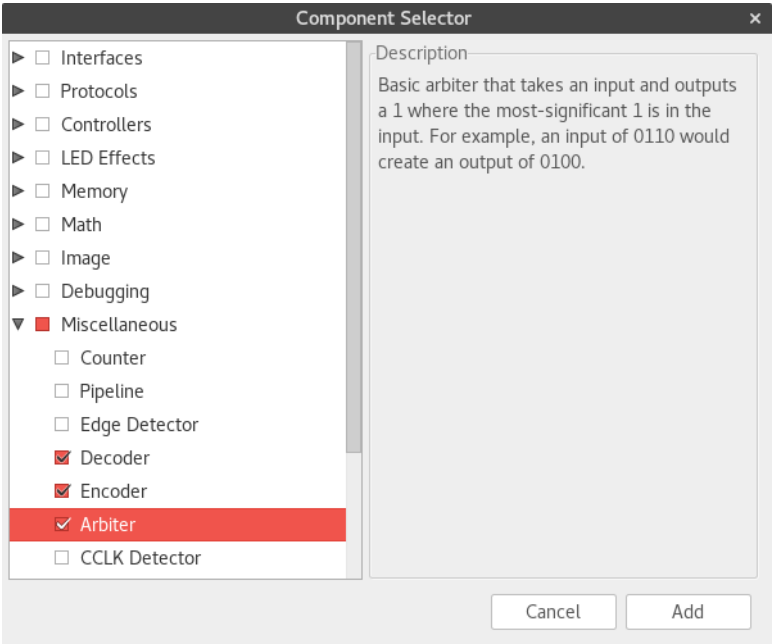

Figure 3-13. Components Library

With those three selected, click Add. They will then show up in your project under the Components branch, as shown in Figure 3-14. You can open and view any of the components' source code to see how they work and for info on how to use them, but they can't be edited.

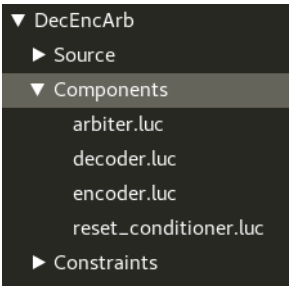

Figure 3-14. Component source tree

With the components added to your project, we need to instantiate them and hook them up to switches and LEDs.

Each of these components has a parameter called `WIDTH` that is used to specify the input width. In the case of a decoder, the output will be 2^{WIDTH}. The encoder's output will be a ceiling of $\log_2(WIDTH)$, and the arbiter's output is just `WIDTH`.

When you declare a module, you can add an optional parameter list. If you want to use parameters, you add `#(PARAM_DEC, PARAM_DEC, ...)` between the name of the module and the port declaration. The actual parameter declaration, `PARAM_DEC`, in its simplest form is simply the parameter name. You can then add a default value by using `NAME = default_value`. This is the value that will be used if the parameter isn't overwritten when the module is instantiated. It's generally a good idea to specify a reasonable default if possible.

Parameter and constant names must be made up of only capital letters and underscores.

Sometimes you will want to constrain the parameter's value. In the case of `WIDTH`, it doesn't make sense for this value to be negative or zero, so we add a constraint by using the *`: EXPRESSION`* syntax. The expression can use the parameter it belongs to or any parameters declared above it. If this value is nonzero, the constraint passes. If the constraint fails, it will throw an error when the module is being instantiated. You should always add constraints to parameters if there are any values that won't work.

The parameter declaration for `WIDTH` in the decoder looks like this:

```
WIDTH : WIDTH > 0 // width of the input
```

In this case, a default wasn't specified because there really isn't a good default value, and every use of the decoder should overwrite it.

We are going to hook all three up to their own block of eight switches and LEDs so the inputs and outputs can be no more than 8 bits wide. We then need to set the `WIDTH` for the decoder to be 3 ($2^3 = 8$) and 8 for the encoder and arbiter.

With that set, we just need to hook them up to the switch and LED groups:

```
module mojo_top (
    input clk,              // 50MHz clock
    input rst_n,            // reset button (active low)
    output led [8],         // 8 user controllable LEDs
    input cclk,             // configuration clock, AVR ready when high
    output spi_miso,        // AVR SPI MISO
    input spi_ss,           // AVR SPI Slave Select
    input spi_mosi,         // AVR SPI MOSI
    input spi_sck,          // AVR SPI Clock
    output spi_channel [4], // AVR general purpose pins
```

```
    input avr_tx,           // AVR TX (FPGA RX)
    output avr_rx,          // AVR RX (FPGA TX)
    input avr_rx_busy,      // AVR RX buffer full
    output io_led [3][8],   // LEDs on IO Shield
    output io_seg [8],      // 7-segment LEDs on IO Shield
    output io_sel [4],      // Digit select on IO Shield
    input io_button [5],    // 5 buttons on IO Shield
    input io_dip [3][8]     // DIP switches on IO Shield
  ) {

  sig rst;                  // reset signal

  .clk(clk) {
    // The reset conditioner is used to synchronize the reset signal to the FPGA
    // clock. This ensures the entire FPGA comes out of reset at the same time.
    reset_conditioner reset_cond;
  }


  decoder dec (#WIDTH(3));
  encoder enc (#WIDTH(8));
  arbiter arb (#WIDTH(8));

  always {
    reset_cond.in = ~rst_n; // input raw inverted reset signal
    rst = reset_cond.out;   // conditioned reset

    led = 8h00;             // turn LEDs off
    spi_miso = bz;          // not using SPI
    spi_channel = bzzzz;    // not using flags
    avr_rx = bz;            // not using serial port

    io_seg = 8hff;          // turn segments off
    io_sel = 4hf;           // select no digits

    dec.in = io_dip[0][2:0];
    enc.in = io_dip[1];
    arb.in = io_dip[2];

    io_led[0] = dec.out;
    io_led[1] = enc.out;
    io_led[2] = arb.out;
  }
}
```

Build and load your project onto the Mojo. If you play with the first (rightmost) group of switches, the LED in the position of the value of the first three switches should light up. For example, if you set the first three switches to 010, or 2, the third LED should turn on (remember, the first LED is 0).

The second group is the encoder. You should set only one switch on at a time, as the input is supposed to be one-hot. However, the way the module is written, it'll just use

the most significant bit if you turn on more than one. It does the exact opposite of the first group of switches. This means if you set the three switches in the first group to a value and then set the switches in the second group to match the LEDs in the first group, the LEDs in the second group should match the switches in the first group. For example, turning on all three switches in the first group will turn on LED 7. Turning on only switch 7 (the leftmost switch) in the second group will cause the three LEDs to turn on, mirroring the three switches in the first group.

The last group of switches is connected to the arbiter. Only the LED that matches up with the leftmost on-switch will light up.

CHAPTER 4
Sequential Logic

In this chapter, we will cover what sequential logic is and, more specifically, what flip-flops are and how to use them. Sequential logic is important for controlling the flow of data through your design as well as improving efficiency by allowing different sections of combinational logic to operate independently. This chapter starts a new example project; you will create a circuit that can keep track of a running total. We will also introduce how to properly read external inputs and solve the challenges that can be associated with them.

It should come as no surprise that because combinational logic is a type of circuit that depends only on the current inputs, the other type, *sequential logic*, is a type of circuit whose output depends on not only the current inputs but also the sequence of previous inputs. This means that the circuit has some sort of memory to be able to keep track of what has happened before.

A basic example is a circuit that keeps a running total of numbers. Each *cycle* (we will define what a cycle is later), it adds the current input to the previous sum, but the previous sum isn't an input. The circuit "remembers" it.

So how do we create circuits that can remember things? It's quite simple: we need to have feedback loops. In the example of the running total, we would need to feed the previous output back as another *internal* input. In practice, it gets a little more complicated than that, though.

Feedback Loops and Flip-Flops

Take a look at the circuit in Figure 4-1.

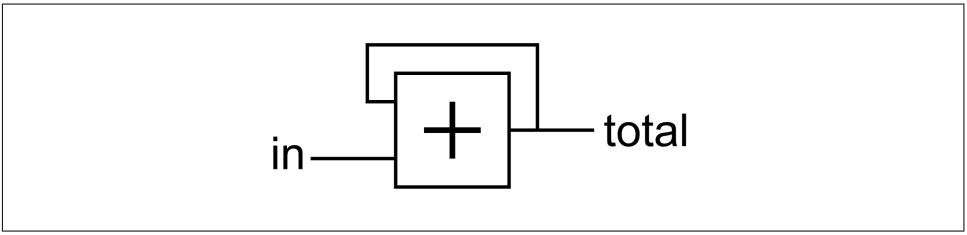

Figure 4-1. Running total circuit

In this diagram, the box with a + in it represents an adder. The two inputs on the left are the two numbers to be added, and the output on the right is the sum. In this example, we loop the sum back to one of the inputs. However, there is a problem. Imagine that somehow we know that `total` starts at 0. If we then change `in` to be 1, what is the output? Well, at first the two inputs are 0 and 1, so the output should obviously be 1. However, when the output is 1, the inputs are then 1 and 1, so the output should be 2. But if the output is 2, the inputs are 2 and 1, so the output should be 3. This will continue on forever.

If we built this circuit, how fast would `total` count up? Could we control the output at all? The speed it would count would be determinated by the incredibly short delay for the values to propagate through the logic gates. In practice, it wouldn't even work. Binary numbers are represented with multiple bits, and the propagation delays for each bit would be different. This would lead to each bit updating its value at slightly different times, giving invalid intermediate values at the input, which would in turn lead to invalid outputs. The invalid outputs would cycle back to the input and produce more garbage. Basically, this is a mess and clearly not the way to keep a running total.

To make this work, we need a way to prevent the feedback from changing immediately. We need to be able to control how often it changes. Luckily, there is a circuit element known as the *D-type flip-flop*, or DFF, that will help us here.

Figure 4-2 is a basic representation of a common DFF. There are three inputs, `D`, `clk`, and `rst`, and a single output, `Q`. Before we get into how these work, we need to cover what a clock is.

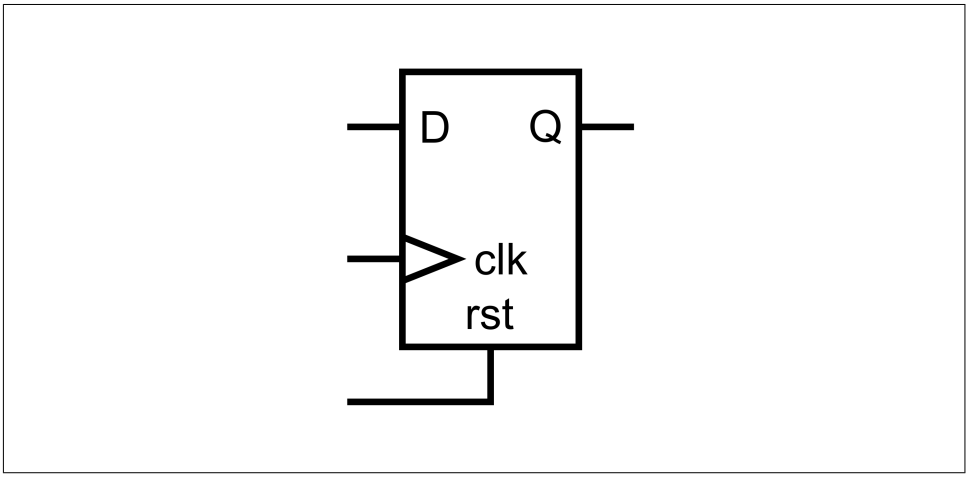

Figure 4-2. D-type flip-flop

Clocks

You have likely heard about clock signals before without even knowing about it. When you buy a computer, the spec sheet will always list the *clock speed* of the processor, something like 2.8 GHz. So what does this mean?

A `clock` is a signal that toggles between 0 and 1 with a set frequency. It looks something like Figure 4-3.

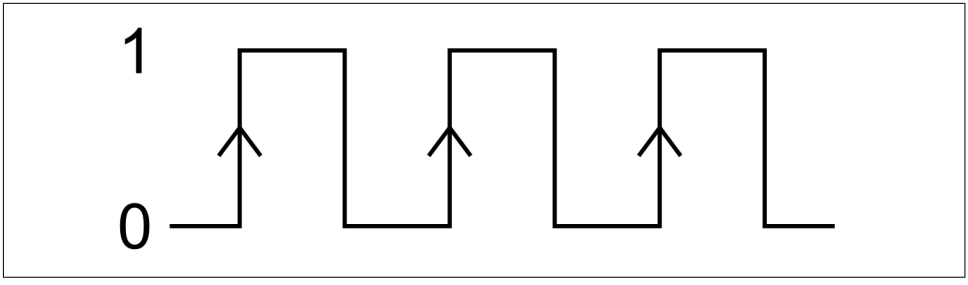

Figure 4-3. Clock signal

The important parts of the clock signal are the edges. There are two types: the rising and falling edges. *Rising edges* occur when the clock transitions from 0 to 1 and are marked with a little up arrow in Figure 4-3. Most of the time, the rising edge is the one you really care about. *Falling edges* occur the when the clock transitions from 1 to 0 and are used less often.

The time between edges is specified by the clock's frequency. In the example of the computer processor running at 2.8 GHz, the clock will toggle 2.8 billion times a second, or once every 357 picoseconds! The clock on the Mojo isn't quite that extreme. It

runs at 50 MHz, meaning it toggles 50 million times a second, or once every 20 nanoseconds.

So back to the DFF. As you may have guessed, the `clk` input stands for *clock*. The function of a DFF is to copy the value from `D` and output it on `Q` on each rising edge of the clock. Between rising edges, if the value at `D` changes, the value at `Q` won't change. `Q` changes only when there is a rising edge. In other words, the flip-flop will remember the last value of `D` and output it until it is told to remember a new value (at the next rising edge of the clock).

The input `rst` stands for *reset* and is used to force the output to a known value. In the case of our running total, we want the initial total to be 0. We can use the reset input to force the total to 0 when the circuit is powered on or needs to be reset. Without a reset, DFFs initialize to a random value.

The reset input usually forces the DFF to 0, but you can also have it *set* the DFF, forcing it to 1.

You will also sometimes see flip-flops drawn with another input called *enable*. This is used to select which rising edges the flip-flop should update on.

With your new knowledge of the DFF, take a look at the new running total circuit in Figure 4-4.

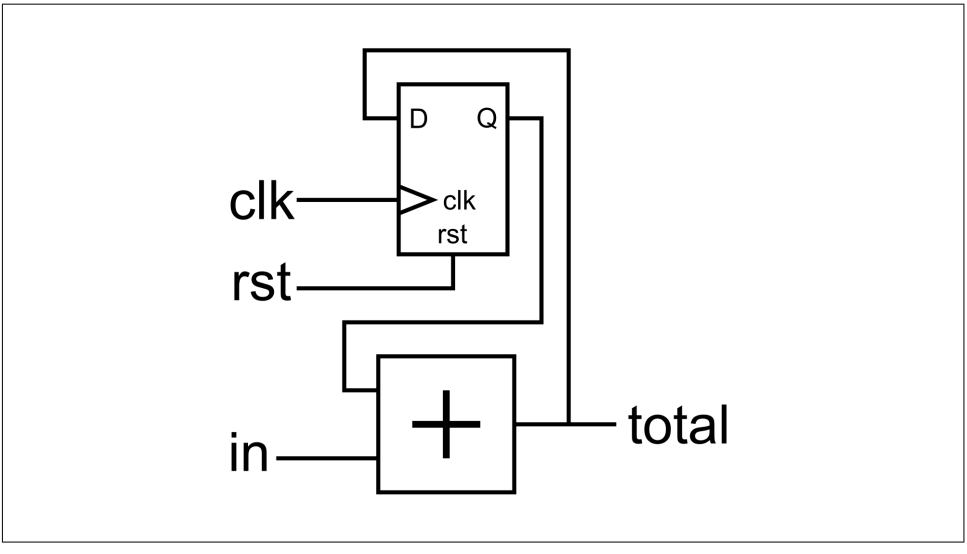

Figure 4-4. Running total circuit with DFF

The new circuit has two more inputs, `clk` and `rst`. The clock will dictate when the next value of `in` is added to `total`, and the reset input will allow us to reset the total to 0.

Now if we were to implement this circuit on the Mojo and use the switches as the input and the LEDs as the output, we couldn't easily control it because it would add the input to the total 50 million times per second. Let's go ahead and do it anyway to get a feel for using DFFs.

Create a new project based on the IO Shield Base and open up *mojo_top.luc*.

In Lucid, we have a type `dff` that we can use to instantiate DFFs in our design. Just as in the diagram, they have three inputs (`clk`, `rst`, and `d`) and one output (`q`). Since most of your design will use the same reset and clock signals, Lucid has a way to easily connect the same signal to many instances of DFFs or other modules:

```
.clk(clk) {
  // The reset conditioner is used to synchronize the reset signal to the FPGA
  // clock. This ensures the entire FPGA comes out of reset at the same time.
  reset_conditioner reset_cond;

  .rst(rst) {
    dff running_total[24]; // Running total, 24 bits wide
  }
}
```

Here we use the *`.port_name(signal_name){}`* syntax to connect `clk` and `rst` to the `clk` and `rst` inputs of the DFF. With this syntax, *`port_name`* is the name of the port on the instance. A port is simply an input, output, or inout of a module. In other words, *`signal_name`* is the name of the signal to connect to the *`port_name`* input of the DFF, in this case. It's a little confusing here because the names of the input and signal are the same.

Also note that we nested the two port connection blocks. The module `reset_cond` won't have `rst` connected because it isn't in the block for that. However, `running_total` will have both `rst` and `clk` connected. You can list multiple connections for a single block by separating them with commas. For example, if we didn't have `reset_conditioner` there, we could have instantiated the DFF as follows:

```
.clk(clk), .rst(rst) {
  dff running_total[24]; // Running total, 24 bits wide
}
```

Note that the port connection block syntax is great when you have a bunch of DFFs in your design, but if you have a single DFF, you could also connect only to that DFF with the following:

```
dff running_total[24] (.clk(clk), .rst(rst)); // Running total, 24 bits wide
```

You can mix and match any of these port connections to fit your needs. However, it is usually a good idea to set up a block that connects the clock and reset because they are used so much.

Also the `rst` input to the `dff` type is optional. If you don't connect it, the `dff` won't have a reset. If you don't need a reset, don't use it, as this will make your design easier for the tools to route. In the case of our running total, we definitely need a reset to be able to get the running total to a known value.

We can complete the design as follows:

```
module mojo_top (
    input clk,              // 50MHz clock
    input rst_n,            // reset button (active low)
    output led [8],         // 8 user controllable LEDs
    input cclk,             // configuration clock, AVR ready when high
    output spi_miso,        // AVR SPI MISO
    input spi_ss,           // AVR SPI Slave Select
    input spi_mosi,         // AVR SPI MOSI
    input spi_sck,          // AVR SPI Clock
    output spi_channel [4], // AVR general purpose pins
    input avr_tx,           // AVR TX (FPGA RX)
    output avr_rx,          // AVR RX (FPGA TX)
    input avr_rx_busy,      // AVR RX buffer full
    output io_led [3][8],   // LEDs on IO Shield
    output io_seg [8],      // 7-segment LEDs on IO Shield
    output io_sel [4],      // Digit select on IO Shield
    input io_button [5],    // 5 buttons on IO Shield
    input io_dip [3][8]     // DIP switches on IO Shield
  ) {

  sig rst;                  // reset signal

  .clk(clk) {
    // The reset conditioner is used to synchronize the reset signal to the FPGA
    // clock. This ensures the entire FPGA comes out of reset at the same time.
    reset_conditioner reset_cond;

    .rst(rst) {
      dff running_total[24]; // Running total, 24 bits wide
    }
  }

  always {
    reset_cond.in = ~rst_n; // input raw inverted reset signal
    rst = reset_cond.out;   // conditioned reset

    led = 8h00;             // turn LEDs off
    spi_miso = bz;          // not using SPI
    spi_channel = bzzzz;    // not using flags
    avr_rx = bz;            // not using serial port

    io_seg = 8hff;          // turn segments off
    io_sel = 4hf;           // select no digits

    running_total.d = running_total.q + io_dip[0];
```

```
    io_led =
      {running_total.q[16+:8], running_total.q[8+:8], running_total.q[0+:8]};
  }
}
```

Take a look at the line where we assign `running_total.d`. To access inputs and outputs of an instance, we use the dot (`.`) syntax. This line assigns the `d` input of `running_total` to the sum of the `q` output and the first eight switches. The next line simply connects the `q` output to the LEDs as we did before with `result`.

If you build and load this project onto your Mojo, when all the DIP switches are off nothing happens. However, the moment you flip the first switch, almost all the LEDs seem solid on except for the last few. This is because it is counting so fast we can't see it. The first few LEDs are even pretty dim because the FPGA can't fully turn on and off the LED that fast.

If you flip the DIP switches back to 0, you'll freeze the counter at a specific value. Try this a few times just to prove to yourself that it is counting. If you turn on a few of the switches, it will count so fast you can't see any of the LEDs blinking.

Timing and Metastability

When thinking about designs, it is easiest to assume that the DFF takes an instantaneous snapshot of its input at the rising edge of a clock. However, that isn't true in practice. The DFF requires that its input doesn't change for a small period of time around the rising edge of the clock.

The time required for it to be stable before the rising edge is known as the *setup time*, and the time after the edge is known as the *hold time*.

During the time around the edge, if the `D` input changes, the value of `Q` is not guaranteed to be valid. However, it gets worse. Not only can `Q` not be the value that `D` was, but it also can do some weird things such as oscillate between 0 and 1 or even get stuck somewhere in the middle. This instability can then propagate through your design, making other DFFs unstable.

In Figure 4-5, `D` is stable through the setup and hold times of the first two edges, so `Q` properly copies the value of `D`. However, on the last edge, `D` changes during the setup time, so the value of `Q` is unknown.

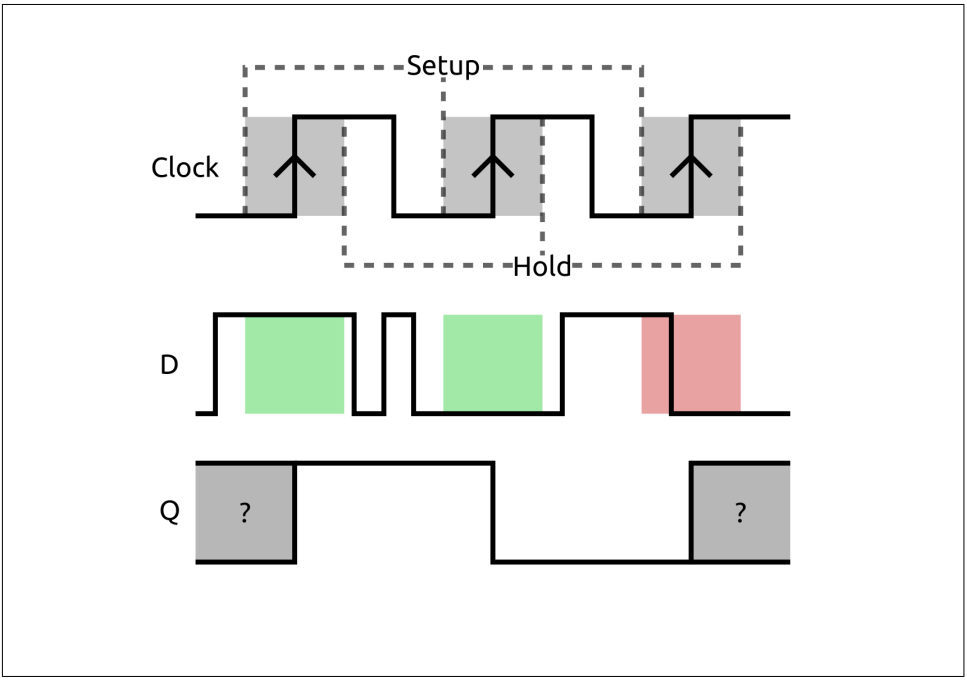

Figure 4-5. Setup and hold times of a DFF. D is valid for the first two edges, but violates the setup time on the last edge.

So how do we make sure that our designs don't violate the flip-flop's timing requirements? For the most part, the tools take care of it. Let's look at an example of two flip-flops connected by an inverter, as shown in Figure 4-6.

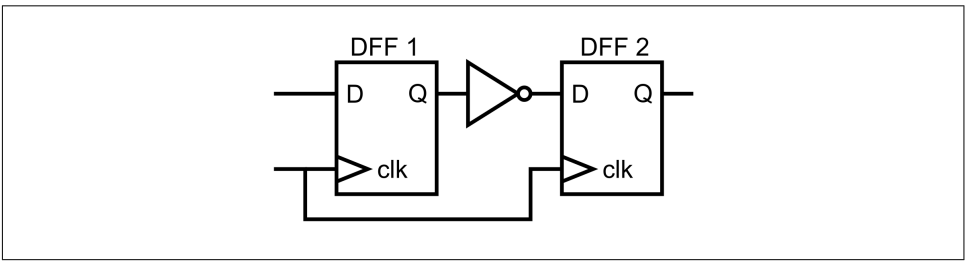

Figure 4-6. Two DFFs connected through an inverter

So what will happen here when `Q` of the first flip-flop is 0, but `D` is 1 and there is a rising edge?

Let's quickly define the terms that you haven't seen yet. If you look back at the schematic for this simple circuit, you'll see that both DFFs have the same clock. It's common for all the DFFs in your entire design to have the same clock. However, the DFFs

are in different locations, and the clock signal originates from a single point. It takes time for the clock signal to reach each DFF. The difference in arrival time of the clock leads to *clock skew*. This means that the rising edges of the clock are not exactly the same for each DFF. Also note that the skew can be both positive and negative, depending how the circuit is actually laid out.

In Figure 4-7, *Clock 1* refers to the clock seen at the input of *DFF 1*, and *Clock 2* refers to the clock seen at *DFF 2*. Both clocks originate from the same source.

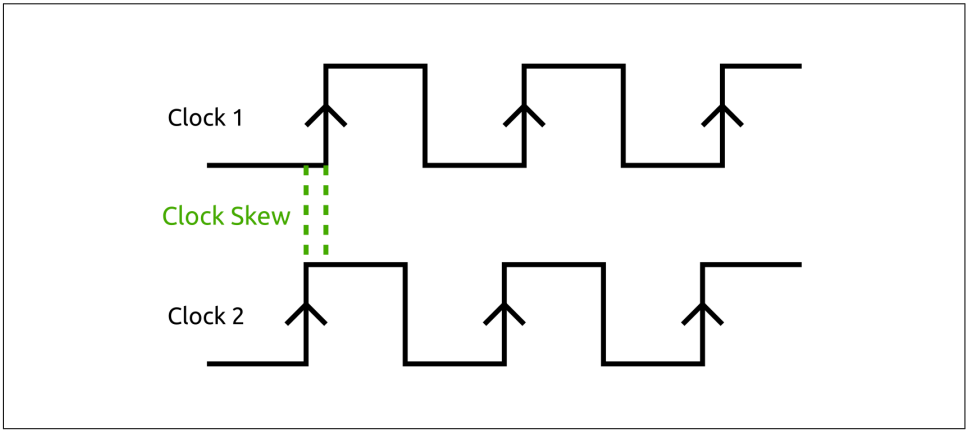

Figure 4-7. Clock skew

Earlier we also assumed that the DFF copied the `D` value to `Q` immediately at the rising edge. However, that's not the case. There is a small amount of time between the rising edge and when `Q` actually changes. This is known as the *clock-to-Q propagation delay*, as shown in Figure 4-8.

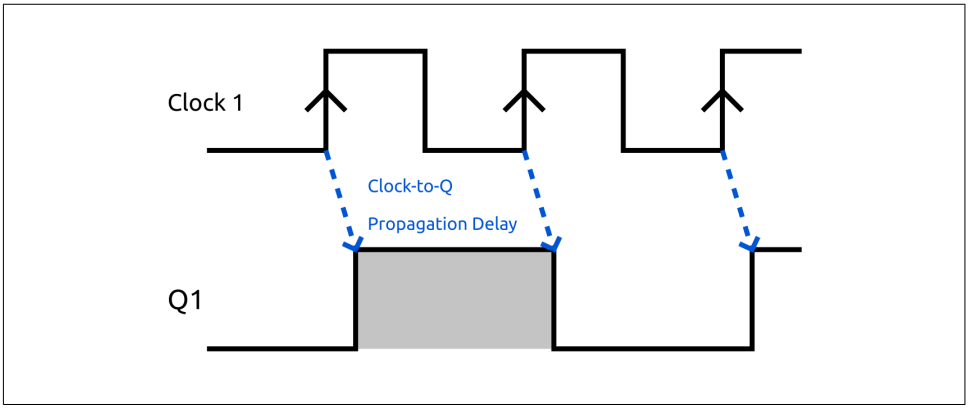

Figure 4-8. Clock-to-Q delay

Once `Q` changes, the new value needs to propagate through the circuit. The time it takes for the old value to be *contaminated* with the new value is called the *contamination delay*.

However, after the contamination delay, the value may toggle a few times before settling on the correct value. The time it takes after `Q` changes to when the output is stable and correct is called the *combinational logic propagation delay*. This delay is a function of your circuit design. The more logic you stick between two flip-flops, the longer this will be. It ultimately dictates the maximum clock speed at which you can run your design. Figure 4-9 illustrates contamination delay and combinational logic propagation delay.

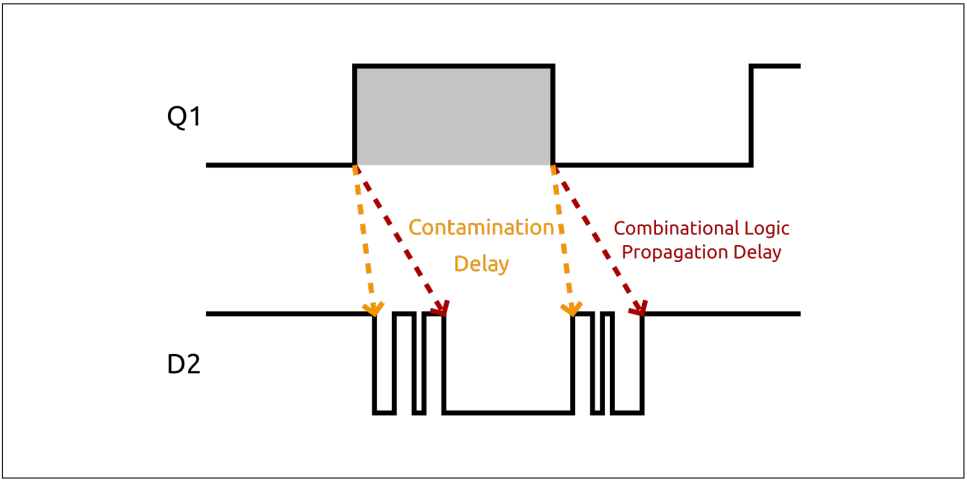

Figure 4-9. Logic delays

If we ignore clock skew, to satisfy hold time we need to make sure that the clock-to-Q delay plus the contamination delay is equal to or greater than the hold time. Note that the clock-to-Q and the contamination delays are a function of how the circuit is laid out and not of the clock frequency. Because of this, we have to fully rely on the tools to make sure hold time is being met.

Again ignoring clock skew, to satisfy setup time, we need to ensure that the period of the clock minus the setup time is greater than or equal to the clock-to-Q delay plus the combinational logic delay. In this case, the period of the clock plays a role in timing. If for some reason the tools can't meet timing on your design, they will prioritize satisfying the hold time over the setup time. This is because you can always satisfy the setup time by reducing your clock frequency (which increases the clock period).

Take a look at all of these constraints put together in Figure 4-10.

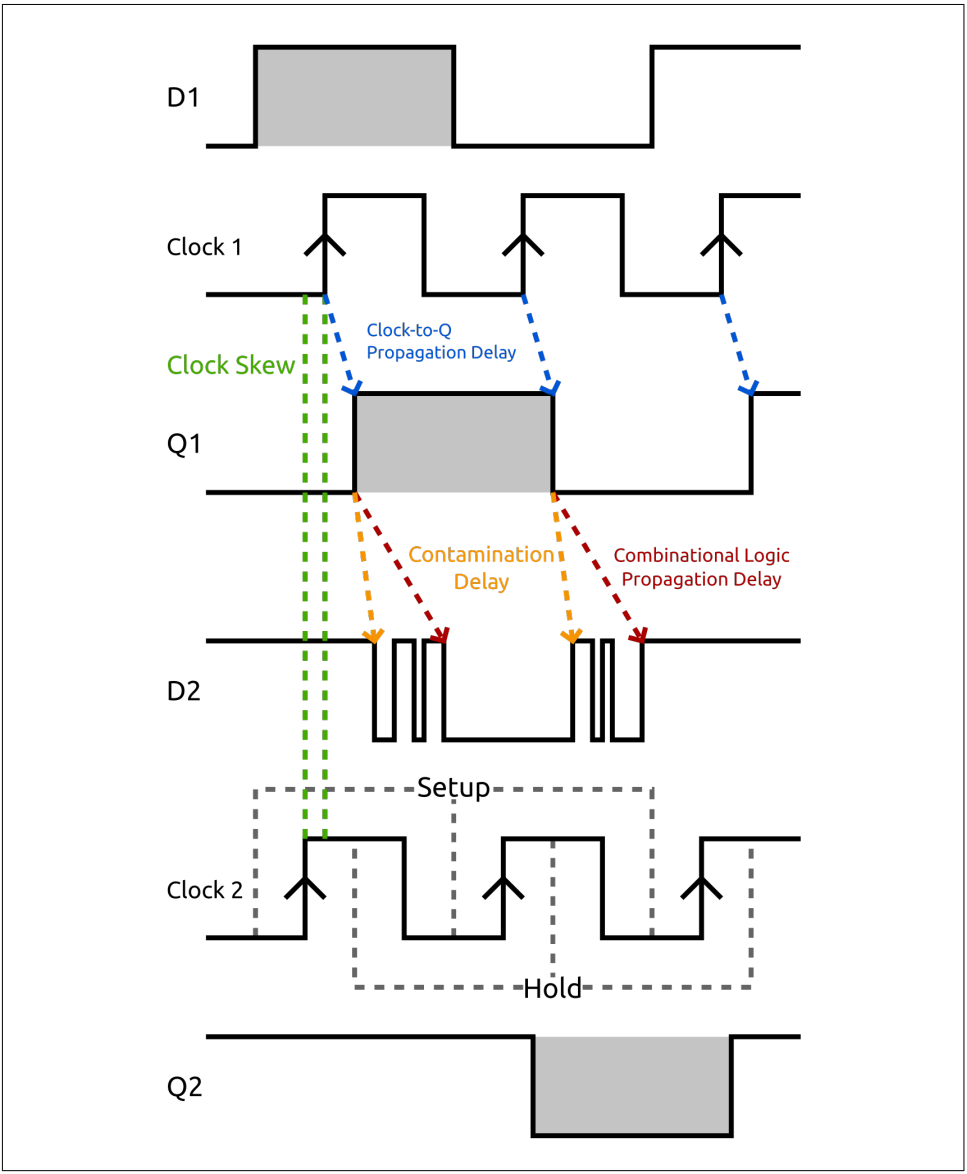

Figure 4-10. Timing of the DFF chain

Notice that in this example the circuit would be functional because the inputs to the flip-flops never change between the setup and hold times around each rising edge.

External Inputs

Luckily for us, the tools will figure out all these timing issues and attempt to lay out your design in such a way that it will all work. However, they can figure out the timing only when things are clearly defined. In some cases, the tools can't satisfy timing for you. One case is when you have external inputs.

You can synchronize some external inputs to the system clock so that you don't have to worry about the following issues. However, some inputs are generated from a different system clock or have no clock at all.

In fact, in our previous example with the counter, we used the DIP switches as an input directly to the counter. What happens if we flip one of the switches right on the rising edge of the clock? There is no way we can possibly synchronize our switch flips. This means we will eventually violate timing and cause stability issues if we flip it enough.

So how do we solve this issue? It is quite simple: we can use a simple circuit known as a *synchronizer*, shown in Figure 4-11.

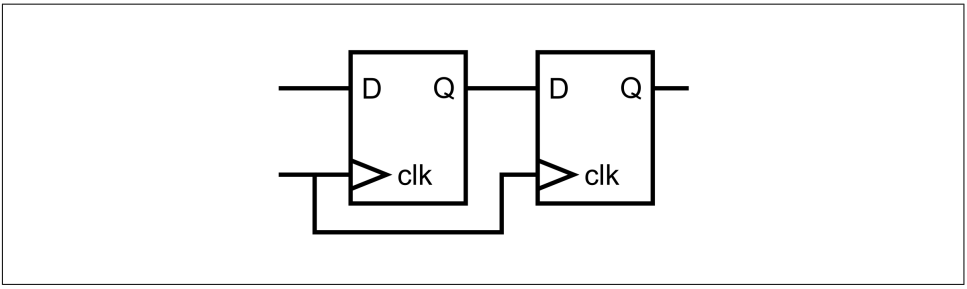

Figure 4-11. Basic synchronizer

The most basic synchronizer is two flip-flops connected together. You feed the unsynchronized input into the leftmost one, and the output on the right will be synchronized to the clock. This works because even if the first flip-flop becomes unstable, it is pretty unlikely that it will also cause the next to become unstable. However, there is still a chance that they both become unstable. To further reduce the possibility of instability, you can chain more than two DFFs together. The more you chain, the lower the probability of instability at the end. However, the payoff quickly diminishes after just two. This is something that all unsynchronized inputs in any digital system suffer from, and there is always that small chance that a value will be read incorrectly, or worse, the instability propagates. However, systems are designed so that their mean time between failures, or MTBF, is ridiculously high. Even just two flip-flops chained like this can have an MTBF of millions of years, depending on the input and flip-flop characteristics.

Add the Components

Open the Components Library, and go to Project → Add Components. We need three of the components for this project: the *Pipeline*, *Edge Detector*, and *Button Conditioner*. All of these can be found under the Miscellaneous category, as shown in Figure 4-12.

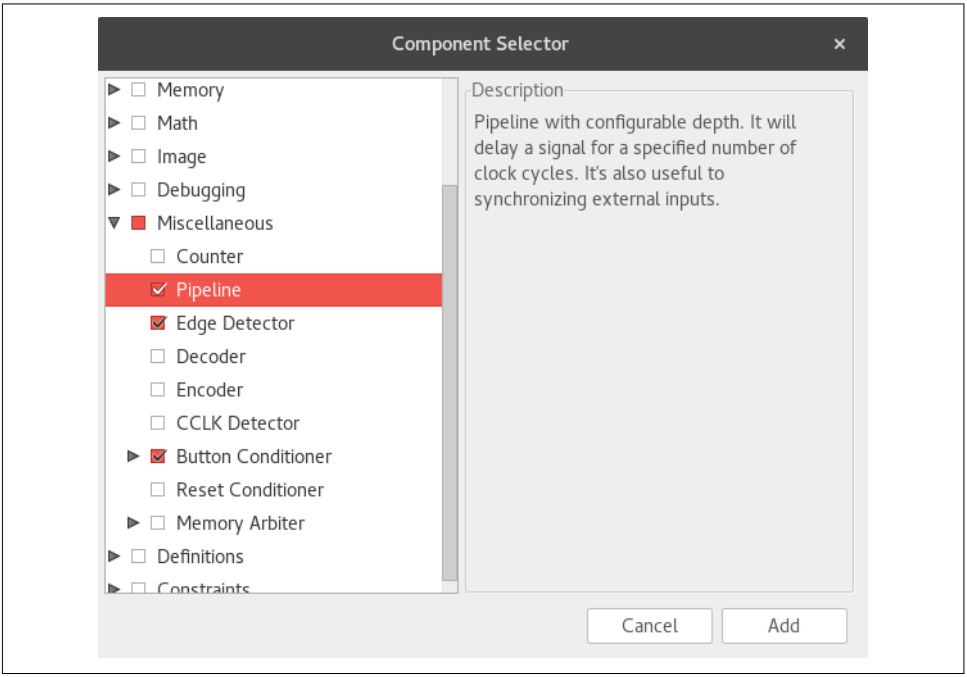

Figure 4-12. Components Library

The *Pipeline* component is simply a chain of DFFs like the synchronizer. The length of the chain is configurable using parameters. Let's take a look at the source:

```
module pipeline #(
    DEPTH = 1 : DEPTH > 0 // number of stages
  )(
    input clk,  // clock
    input in,   // input
    output out  // output
  ) {

  dff pipe[DEPTH] (.clk(clk));
  var i;

  always {
    // in goes into the start of the pipe
    pipe.d[0] = in;
```

```
    // out is the end of the pipe
    out = pipe.q[pipe.WIDTH-1];

    // for each intermediate state
    for (i = 1; i < DEPTH; i++)
      pipe.d[i] = pipe.q[i-1]; // copy from previous
  }
}
```

The parameter `DEPTH` is used as the size of the array of DFFs, `pipe`. Even though `DEPTH` can take different values, it is constant at synthesis time, so you can use it as array sizes and indices.

Looking in the `always` block, the first line connects the input, `in`, to the first `dff` in the chain. The second line connects the output, `out`, to the output of the last flip-flop. To index the last flip-flop, we use the `WIDTH` attribute of the array. The `WIDTH` attribute for a single-dimensional array is simply a number. In this case, it will be the same value as `DEPTH`. For multidimensional arrays, `WIDTH` will be a single-dimensional array with each entry being the width of a different dimension. For example, `dff array[3][5]` will have a `WIDTH` attribute of `{5, 3}`, so that `array.WIDTH[0]` is equal to 3, and `array.WIDTH[1]` is 5.

There is a new type used in this module too, `var`. This stands for *variable* and is used to store a value that won't really be in the hardware design but is useful in the description. The most common use for a `var` is as the index of a `for` loop.

Now let's take a look at the `for` loop:

```
for (i = 1; i < DEPTH; i++)
  pipe.d[i] = pipe.q[i-1]; // copy from previous
```

`For` loops take a similar form to C/C++ or Java for loops. They have the syntax `for (`*`init_expr`*`; `*`condition_expr`*`; `*`operation`*`) `*`statement`*. Only the type `var` should be used in `for` loops. The *`init_expr`* sets the initial value of the `var`—in this case, 1. The *`condition_expr`* is evaluated before each iteration to see whether the loop should continue. Finally, *`operation`* specifies how the `var` will change on each iteration.

Loops in Lucid my seem similar to software for loops you might be familiar with. However, they are implemented differently. `for` loops should be thought of as shorthand for writing everything out. The synthesizer will essentially unroll the `for` loop, duplicating the hardware required for each iteration. Unlike code, they don't make your design any smaller. They only make the design files smaller and easier to maintain.

In this case, we couldn't unroll the loop by hand because the number of iterations is specified by the parameter `DEPTH`. However, `DEPTH` is a constant (although it may have different values for different instances), so the tools can unroll it. `for` loops can be

used only when they have a constant number of iterations that can be determined at synthesis time.

This `for` loop will simply connect the output of the previous flip-flop to the input of the next, creating the chain. Note that if `DEPTH` is 1, which is allowed with our constraint, the `for` loop will be ignored because the *`condition_expr`* will fail on the first iteration. This is what we want, because the input and output of the single `dff` would already be connected to the module's ports.

Take a look at Figure 4-13 to see how this will look in hardware.

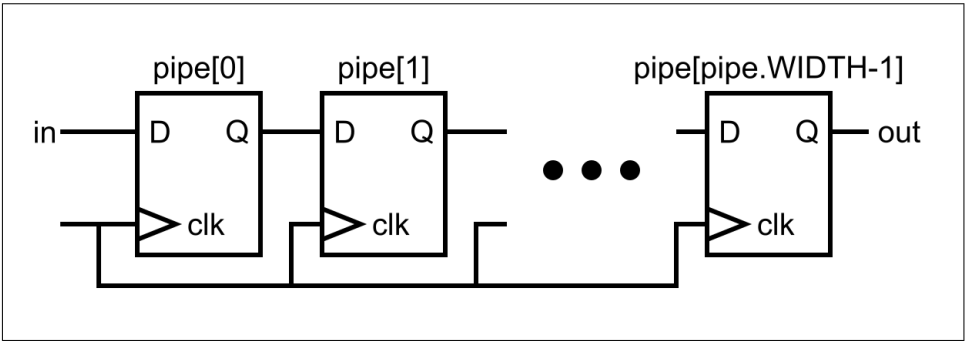

Figure 4-13. Hardware pipeline

Instantiating a Module

Now that you have a solid understanding of the pipeline component, let's use it in the running total project to synchronize the switch inputs with the clock:

```
module mojo_top (
    input clk,              // 50MHz clock
    input rst_n,            // reset button (active low)
    output led [8],         // 8 user controllable LEDs
    input cclk,             // configuration clock, AVR ready when high
    output spi_miso,        // AVR SPI MISO
    input spi_ss,           // AVR SPI Slave Select
    input spi_mosi,         // AVR SPI MOSI
    input spi_sck,          // AVR SPI Clock
    output spi_channel [4], // AVR general purpose pins
    input avr_tx,           // AVR TX (FPGA RX)
    output avr_rx,          // AVR RX (FPGA TX)
    input avr_rx_busy,      // AVR RX buffer full
    output io_led [3][8],   // LEDs on IO Shield
    output io_seg [8],      // 7-segment LEDs on IO Shield
    output io_sel [4],      // Digit select on IO Shield
    input io_button [5],    // 5 buttons on IO Shield
    input io_dip [3][8]     // DIP switches on IO Shield
  ) {
```

```
    sig rst;                  // reset signal

    .clk(clk) {
      // The reset conditioner is used to synchronize the reset signal to the FPGA
      // clock. This ensures the entire FPGA comes out of reset at the same time.
      reset_conditioner reset_cond;

      // create a 3x8 array of pipelines with a depth of 2
      pipeline switch_cond[3][8] (#DEPTH(2));

      .rst(rst) {
        dff running_total[24]; // Running total, 24 bits wide
      }
    }

    always {
      reset_cond.in = ~rst_n; // input raw inverted reset signal
      rst = reset_cond.out;   // conditioned reset

      led = 8h00;             // turn LEDs off
      spi_miso = bz;          // not using SPI
      spi_channel = bzzzz;    // not using flags
      avr_rx = bz;            // not using serial port

      io_seg = 8hff;          // turn segments off
      io_sel = 4hf;           // select no digits

      switch_cond.in = io_dip; // connect the raw switch input to the conditioner

      // use the conditioned switch input
      running_total.d = running_total.q + switch_cond.out[0];
      io_led =
        {running_total.q[16+:8], running_total.q[8+:8], running_total.q[0+:8]};
    }
  }
```

To instantiate the module, we use its name followed by the name of our instance. Note that the pipeline module has a `clk` input but not a `rst` input. This means we can put in the port connection block for `clk`, but not with `running_total` in the `rst` port connection block.

We also need the pipeline to be an array of pipelines. To do this, we can tack on the array size to the name. This will duplicate the pipeline module for each element in the array. Because the entire array is in the `clk` port connection block, each individual element will have its `clk` input connected to `clk`:

```
    .clk(clk) {
      ...
      // create a 3x8 array of pipelines with a depth of 2
      pipeline switch_cond[3][8] (#DEPTH(2));
```

```
    ...
}
```

If you instantiate an array of modules and don't connect a port at instantiation, that port will be packed into an array. This is what happens to `switch_cond.in` and `switch_cond.out`. While they are typically a single bit, because `switch_cond` is an array, they become 3 x 8 arrays themselves. This is convenient because we can then directly assign `io_dip` to `switch_cond.in` since they have the same dimensions. This also makes `switch_cond.out` a direct drop-in replacement for `io_dip`.

The default depth of the pipeline is 1, which is pretty useless because you could just use a DFF directly. We need to override this so that we get a chain of two DFFs. To specify a parameter's value, you specify it basically the same way as you do a port connection, except instead of leading the name with a period (.), you use `#`. You can even add parameter values to port connection blocks if everything in the block has the same parameter name. If we didn't instantiate the pipeline module in the port connection block, we could use the following:

```
pipeline switch_cond[3][8] (#DEPTH(2), .clk(clk));
```

The order of parameters and port connections don't matter, but typically parameters are assigned first.

If you try changing `DEPTH` to an invalid value such as 0, you will get an error from the failing constraint:

```
Line 28, Column 13 : The constraint "DEPTH>0" for parameter "DEPTH" with
value "0: {0}" failed
```

If you build and load this project, it should behave basically the same as before—except this time we don't have to worry about metastability issues.

Checking Timing

How do we know that the tools were able to meet the timing requirements? If you look at the output when you build your project, a little bit above the end there is a section that looks something like this:

```
------------------------------------------------------------------------
  Constraint   | Check | Worst Case |  Best Case | Timing |   Timing
               |       |    Slack   | Achievable | Errors |    Score
------------------------------------------------------------------------
  TS_clk =     | SETUP |    14.461ns|     5.539ns|       0|           0
  PERIOD
  TIMEGRP
  "clk" 50
  MHz HIGH
   50%         | HOLD  |     0.447ns|            |       0|           0
------------------------------------------------------------------------
```

```
All constraints were met.
```

The important line here is the `All constrains were met.` part. This means the tools were able to route our design so that it should work under all operating conditions that the FPGA is designed to work in. If there was a problem, the Timing Errors and Timing Score columns would have nonzero values.

We will cover later what to do if timing fails, but for most designs running at 50 MHz, timing won't be an issue. In the preceding case, the Best Case Achievable time of 5.539 ns means this design could run at 180 MHz (1/5.539 ns) without problems.

Also how do the tools know that the clock is 50 MHz? We told them! If you look under the Constraints section of the project tree, you will find a file called *mojo.ucf*. This file is where we tell the tools what pins on the FPGA to use for each signal internal to the Mojo as well as the frequency of the clock. The file *io_shield.ucf* contains the pin constraints used by the IO Shield.

Don't worry about the *user constraints file* (UCF) details for now: we will get to them later.

Bouncy Buttons and Edges

We are going to modify the project further so that it adds the switch value only when we push a button. But first, you need to know some stuff about buttons. We could try to synchronize the button input as we did with the switches, but buttons have another issue: they often bounce when you press them. The contacts, instead of staying closed, hit each other and can then separate before slamming back together, as shown in Figure 4-14. This can happen a few times or none at all. To compensate for this, we need to register button presses only when they have been pressed and continue to be pressed for a small amount of time.

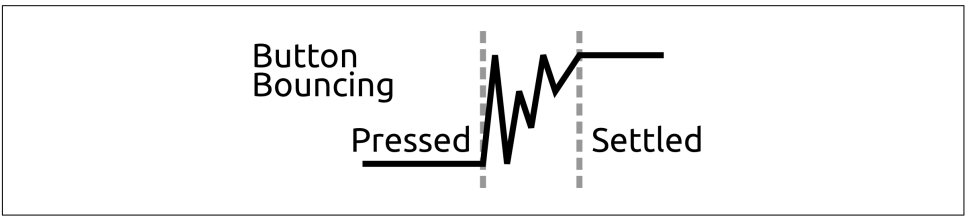

Figure 4-14. Button bouncing

Luckily, the Button Conditioner component takes care of this for us.

Feel free to take a look at the contents of the module located in the *button_conditioner.luc* file. It works by first passing the button input through a pipeline, as we did with the switches. If the synchronized output is 1 (the button is assumed active high), a counter is incremented until it reaches its maximum value. When it reaches the

maximum value, the button conditioner outputs 1. If at any point the button input is 0, the counter is reset and the output of the conditioner is 0.

The button conditioner will output 1 when the button is pressed, and 0 when it isn't. However, we want to add the number only once for each time the button is pressed. To do this, we need to be able to detect when the button goes from 0 to 1. This is easy to do with a single DFF. You simply feed the value into the DFF. The output of the DFF will then be the value of the button from the last clock cycle. If the button was 0 on the last clock cycle, but is now 1, we know it was just pressed.

However, we don't need to do this because another component does this for us: the *Edge Detector* component.

If you take a look at its source, you'll see it does basically what was just described. The one main difference is that you can configure it so that it will detect rising, falling, or both types of edges using parameters.

Running Total

We now have all the pieces we need to make a more useful running total design.

First, we need to instantiate the button conditioner and edge detectors:

```
.clk(clk) {
  // The reset conditioner is used to synchronize the reset signal to the FPGA
  // clock. This ensures the entire FPGA comes out of reset at the same time.
  reset_conditioner reset_cond;

  // create a 3x8 array of pipelines with a depth of 2
  pipeline switch_cond[3][8] (#DEPTH(2));
  button_conditioner btn_cond;
  edge_detector edge (#RISE(1), #FALL(0));

  .rst(rst) {
    dff running_total[24]; // Running total, 24 bits wide
  }
}
```

Again, both of these don't have `rst` inputs, so we can stick them in the `clk` port connection block. The default parameter values for the button conditioner are perfect, so we don't need to override them. However, the edge detector needs to be told to look for only rising edges.

Finally, we need to connect it all up in the `always` block:

```
always {
  reset_cond.in = ~rst_n; // input raw inverted reset signal
  rst = reset_cond.out;   // conditioned reset

  led = 8h00;             // turn LEDs off
  spi_miso = bz;          // not using SPI
```

```
    spi_channel = bzzzz;    // not using flags
    avr_rx = bz;            // not using serial port

    io_seg = 8hff;          // turn segments off
    io_sel = 4hf;           // select no digits

    switch_cond.in = io_dip; // connect the raw switch input to the conditioner

    btn_cond.in = io_button[4]; // io_button[4] is the "right" button
    edge.in = btn_cond.out;     // connect the button to the edge detector

    if (edge.out) // only add when the button is pressed
      // use the conditioned switch input
      running_total.d = running_total.q + switch_cond.out[0];

    io_led =
      {running_total.q[16+:8], running_total.q[8+:8], running_total.q[0+:8]};
  }
```

By using the `if` statement, we are able to assign `running_total.d` the new sum only when `edge.out` is 1. You may remember from before that under all circumstances, you need to assign a signal a value. The one exception to this rule is the `d` input of the DFF. This is because a DFF is able to save its last value, while a basic wire can't. If you don't assign it a value, the `q` output will stay the same.

Build and load the project onto your Mojo. By setting the DIP switches and pressing the "right" button, you can now add individual numbers to the running total. If you want to reset the total back to 0, press the button on the Mojo itself (not on the IO Shield).

Congratulations on completing the working running total design! This design is not only functional, but also quite robust, with all the proper precautions taken to clean up the external signals. You can try to connect `io_button[4]` to `edge.in` directly to see what kind of bad behavior you get when you use the external input directly. In my testing, the button press would be ignored about every third press. This obviously isn't an issue of bouncing (which would cause multiple additions per press), so I must be violating timing and causing stability issues. A tricky thing with timing violations is that the behavior it can produce can seem random and unrelated to the input causing it (as the instability can propagate).

CHAPTER 5

Seven-Segment LED Displays and Finite-State Machines

In this chapter, we will create a new project that will teach you how to use the seven-segment displays on the IO Shield, shown in Figure 5-1. Seven-segment displays are common, and you have likely seen them all over the place (for example, on microwave ovens). We will continue building on this project to create a stopwatch that will serve as an example for a finite-state machine.

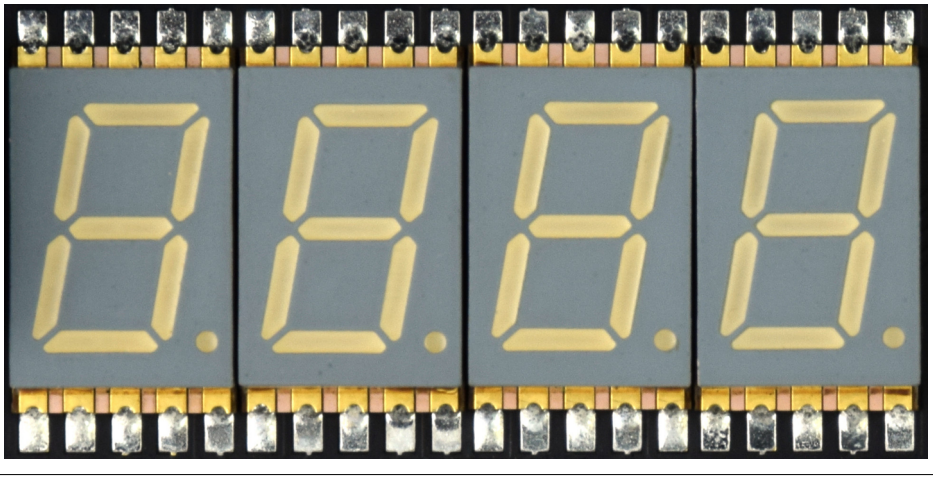

Figure 5-1. Seven-segment displays on the IO Shield

Finite-state machines, or FSMs, are an important design idea that will allow you to create designs with complex behavior. The basic idea is that your design has a finite set of states that it can be in, and various inputs will cause it to transition between

states. The classic example of an FSM is a traffic light. The light can be in a few states: green in one direction, red in both directions, green in the other direction, and so forth. Depending on where cars are detected, it will transition between these states in a specific way.

Single Digit

On the IO Shield, each segment from a digit is connected to the same segment of the other three digits. Given this, it seems as if all four digits would always display the same thing. However, each digit can be turned on and off individually. This is accomplished by connecting the common pin of each digit to a transistor. By turning the transistors on and off, we can enable and disable each digit. By quickly cycling through the digits, we can create the illusion that all the digits are on and that they are all displaying unique numbers. This technique is known as *multiplexing*.

Before we get ahead of ourselves, let's figure out how to display a single digit on just one digit. As before, create a new project based on the IO Shield Base example project.

In the `mojo_top` module, we have two relevant outputs: `io_seg` and `io_sel`. The output `io_seg` connects to the eight LEDs in the digit (the seven segments plus a decimal point). The output `io_sel` connects to the transistors that turn the digits on and off. Both of these signals are active low. That means a 0 value will turn on the LED or digit.

To get started, we can display a static value by changing the values of these outputs. Take a look at the last two lines in the `always` block:

```
io_seg = 8hff;          // turn segments off
io_sel = 4hf;           // select no digits
```

We can edit these values to display a 2 on the first digit:

```
io_seg = 8b10100100;    // "2"
io_sel = 4b1110;        // select first digit
```

If you build and load the project, the first digit should turn on and display a 2. So how did we know what value to set `io_seg` to? Figure 5-2 shows which segment connects to which bit. Remember that both of these signals are active low, so a value of 0 will enable the digit or turn on the LED segment.

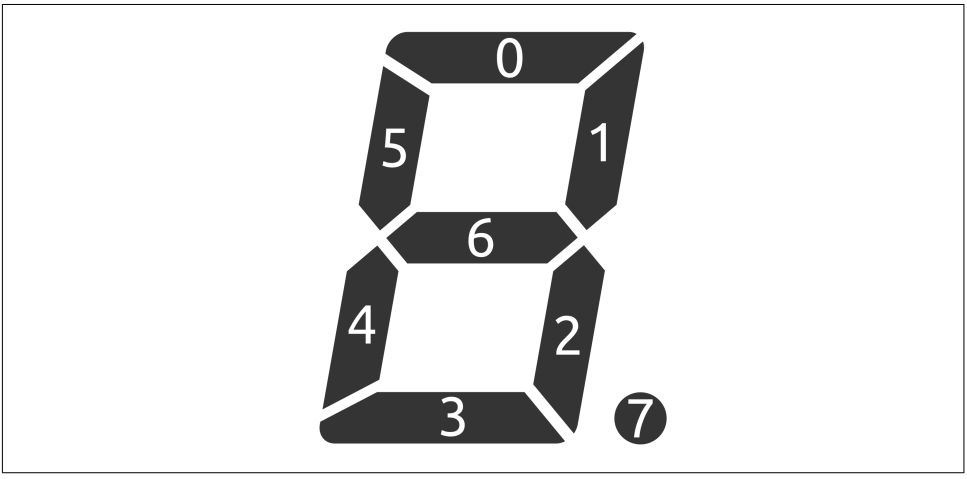

Figure 5-2. LED segment mapping

We can use a counter and a decoder to index through the LEDs as well. Add the *Counter* and *Decoder* components to your project. They can be found under the Miscellaneous category in the Components Library.

Then instantiate the counter and decoder. Note that the decoder is strictly combinational logic and doesn't have a clock or reset input, while the counter has both:

```
.clk(clk) {
  // The reset conditioner is used to synchronize the reset signal to the FPGA
  // clock. This ensures the entire FPGA comes out of reset at the same time.
  reset_conditioner reset_cond;

  .rst(rst) {
    counter ctr (#SIZE(3), #DIV(24));
  }
}

decoder num_to_seg (#WIDTH(3));
```

The Counter component has a handful of parameters that make it flexible. In this case, we are using `SIZE` and `DIV`. The parameter `SIZE` specifies the width of the output. In our case, we want 3 bits so that the counter will count from 0 to 7. The `DIV` parameter specifies the number of bits to use as a pre-counter to divide the clock. By specifying 24 here, the counter will increment its output every 2^{24}, or 16,777,216, clock cycles. With a clock frequency of 50 MHz, this means it will change about every third of a second. Finally, we need the decoder to turn the binary value to a one-hot value to turn on the LED segment. By setting `WIDTH` to 3 bits, we get an output of 8 bits, which is exactly what we want!

All that's left to do is to connect the counter to the decoder and the decoder to the LEDs:

```
num_to_seg.in = ctr.value; // connect counter to decoder

io_seg = ~num_to_seg.out;  // connect decoder to LEDs
io_sel = 4b1110;           // select first digit
```

Remember that the LEDs are active low and the decoder output is one-hot. We really need it to be one-cold. We can get that by inverting the bits.

If you build and load the project now, each segment should cycle through. You can adjust the speed by changing the `DIV` parameter on the counter. A higher value will result in a slower cycle.

What if we want to also use other digits? We can modify the counter and add a second decoder to cycle the `io_sel` output too:

```
.clk(clk) {
  // The reset conditioner is used to synchronize the reset signal to the FPGA
  // clock. This ensures the entire FPGA comes out of reset at the same time.
  reset_conditioner reset_cond;

  .rst(rst) {
    counter ctr (#SIZE(5), #DIV(24));
  }
}

decoder num_to_seg (#WIDTH(3));
decoder num_to_digit (#WIDTH(2));
```

Notice that we made the counter's output 5 bits wide and added another decoder with a 2-bit input. We are going to use the 2 extra bits to feed into the new decoder, which we will connect to the *io_sel* signal. This way, each time the single digit has gone through a full cycle, we will select a new digit for the next cycle:

```
num_to_seg.in = ctr.value[2:0];   // connect counter to segment decoder
num_to_digit.in = ctr.value[4:3]; // connect counter to digit decoder

io_seg = ~num_to_seg.out;         // connect decoder to LEDs
io_sel = ~num_to_digit.out;       // connect decoder to select
```

If you build and load the project this time, after a complete cycle, the lit LED should jump to the next digit.

You should now have a decent understanding of how the seven-segment LED displays work. We can now get a little fancier and start displaying useful values.

Lookup Tables

An important technique in hardware design is using *lookup tables*, also known as *LUTs*. These are tables from which you can select one of many constant values. You can think of a lookup table as an arbitrary function mapping a range of inputs to whatever output values you want. We are going to use one to convert a binary number to the LEDs that should be lit.

Let's put the digit LUT in its own module so we can reuse it. To create a new module, click the little document icon in the toolbar and name the file *digit_lut.luc* and select Lucid Source. The Mojo IDE will automatically populate the file with a bare-bones module:

```
module digit_lut (
    input clk,  // clock
    input rst,  // reset
    output out
  ) {

  always {
    out = 0;
  }
}
```

We are going to make this fully combinational so we can remove the clock and reset inputs and replace them with a `value` input. We also need to replace the `out` output with a `segs` output that is 7 bits wide.

In the `always` block, we can remove the default assignment and replace it with a `case` statement that will do the lookup for us:

```
module digit_lut (
    input value [4],
    output segs [7]
  ) {

  always {
    case (value) {
      0: segs = 7b0111111;
      1: segs = 7b0000110;
      2: segs = 7b1011011;
      3: segs = 7b1001111;
      4: segs = 7b1100110;
      5: segs = 7b1101101;
      6: segs = 7b1111101;
      7: segs = 7b0000111;
      8: segs = 7b1111111;
      9: segs = 7b1100111;
      default: segs = 7b0000000;
    }
```

```
    }
  }
```

A `case` statement works by looking at the value in the expression after the `case` keyword and then using the branch with that value. For example, when `value` is 2, the line *segs = 7b1011011;* will be used. It is important to assign a value to *segs* no matter what value `value` is. That is where the `default` entry comes in. This statement will be used if `value` doesn't match any of the other values.

The values used here are active high, meaning a 1 should turn on the LED. We will need to invert these before we connect them to the actual LED outputs.

A `case` statement is identical to using a bunch of `if-else` statements. It is more compact and easier to write out in some situations. Unlike programming, there is no performance benefit to it.

We can test out the `digit_lut` module by connecting a new counter to it and connecting it to the LEDs in *mojo_top.luc*. We no longer need the decoders and the old counter, so we can remove them:

```
.clk(clk) {
  // The reset conditioner is used to synchronize the reset signal to the FPGA
  // clock. This ensures the entire FPGA comes out of reset at the same time.
  reset_conditioner reset_cond;

  .rst(rst) {
    counter ctr (#SIZE(4), #DIV(24), #TOP(9));
  }
}

digit_lut lut;
```

We also changed `SIZE` down to 4 and set a new parameter, `TOP`, to 9. The parameter `TOP` sets the maximum value for the counter. If you don't set it, it will overflow naturally. However, with 4 bits, it would count from 0 to 15, and we don't have a digit to display from 10 to 15, so we really want it to count only from 0 to 9:

```
  lut.value = ctr.value; // connect counter to LUT

  io_seg = ~lut.segs;    // connect LUT to LEDs
  io_sel = 4b1110;       // select first digit
```

We can connect everything, but remember that the LUT was active high, so we inverted its output.

Building and loading the project should make the first digit continuously count from 0 to 9. We now need to create a way to get all the digits showing a number.

To do this, we will create another module, `multi_seven_seg`. This module will take a value for each digit and display it on the respective digit. It will do this by cycling though them faster than we can see, making them all look like they are on. For this module, we will assume that the LEDs and select signals are active high and invert them when we connect it to the output.

Also our `digit_lut` module assumes that we have only seven segments. However, we really have eight per digit because of the decimal place. The decimal place is the MSB of the output `seg`. To allow these to be used, we will accept an array that will specify which decimal places should be lit. We can concatenate the corresponding bit to the beginning of the `digit_dec` output to get the full eight segments:

```
module multi_seven_seg #(
    DIGITS = 4 : DIGITS > 0,
    DIV = 16 : DIV >= 0
  )(
    input clk,                // clock
    input rst,                // reset
    input values [DIGITS][4], // values to show
    input decimal [DIGITS],   // decimal points
    output seg [8],           // LED segments
    output sel [DIGITS]       // Digit select
  ) {

  // number of bits required to store DIGITS-1
  const DIGIT_BITS = $clog2(DIGITS);

  .clk(clk), .rst(rst) {
    counter ctr (#DIV(DIV), #SIZE(DIGIT_BITS), #TOP(DIGITS-1));
  }

  digit_lut seg_dec;                          // segment decoder
  decoder digit_dec (#WIDTH(DIGIT_BITS));     // digit decoder

  always {
    seg_dec.value = values[ctr.value];        // select value for active digit
    seg = c{decimal[ctr.value], seg_dec.segs}; // output the decoded value

    digit_dec.in = ctr.value;                 // decode active digit to one-hot
    sel = digit_dec.out;                      // output the active digit
  }
}
```

Instead of hardcoding four digits, this module uses the parameter `DIGITS`. We also break out the `DIV` parameter from the counter so we can adjust how fast it switches between digits. The faster we switch, the more power is used, and the LEDs will

appear a little less bright. However, if we switch too slowly, we will see a flicker. The default value should typically be a solid balance.

This module also uses a constant, `DIGIT_BITS`. Constants are a lot like parameters, but they can be set only inside the module. Their value must be known at synthesis time, and their name can consist only of capital letters and underscores.

In the constant declaration, we also used a function. Functions take the form of `$name(arg, arg, arg, ...)`. Some functions can be used on any signal or value, while others, such as `$clog2`, can be used only on constant values. The function `$clog2` stands for a ceiling of log base 2. In this case, we are using it to determine the number of bits we need to store a value up to `DIGITS-1`.

We use a counter and decoder to generate the select signal as before. However, this time, instead of using the counter and decoder to generate the LED segment value too, we use the LUT from before. The value we feed into the LUT is the input to the module indexed by the currently selected digit. Notice that the input `values` is a 2D array, so `values[0]` is the value to show on the first digit, and `values[1]` is the value for the second digit.

In `mojo_top`, we can remove the counter and LUT from before and instantiate the new module:

```
.clk(clk) {
  // The reset conditioner is used to synchronize the reset signal to the FPGA
  // clock. This ensures the entire FPGA comes out of reset at the same time.
  reset_conditioner reset_cond;

  .rst(rst) {
    multi_seven_seg seg_display;
  }
}
```

Then connect it up to the actual LEDs. For testing, we can just use some constant `values`:

```
  seg_display.values = {4d1, 4d2, 4d3, 4d4}; // display 1234
  seg_display.decimal = 4b0000;              // no decimal points

  io_seg = ~seg_display.seg; // module assumes active high so invert
  io_sel = ~seg_display.sel;
```

Putting this on the Mojo should make all four digits show a different number at the same time. Notice that the LEDs are a bit dimmer than before. This is because each digit is on only one-quarter of the time. If you fix your eyes on something and then shake the board, you should be able to tell they are flickering by seeing individual digits instead of a smooth streak. Note that having `DIV` set to 16 makes it change really fast, so if you can't tell it is flickering, try making this 17 or 18.

Binary to Decimal

We now have a way to display four digits on the LEDs, but we need to specify which digit we want to display on each separately. It would be nice to be able to specify a number that would get displayed across all four digits. To do this, we need to convert the binary number from the DIP switches into a decimal number.

Technically, the number will still be in binary since that is all we have to work with, but we will produce a separate number for each digit from a single binary number.

Luckily, there is a component to do this for us. Go to the Components Library and add the *Binary to Decimal* component, which can be found in the Miscellaneous section.

This component will output the decimal numbers, assuming our input is valid. It uses the special value of 10 to indicate the digit should be blank (if there shouldn't be leading zeros), and the value of 11 to indicate that there was overflow. It is a bit complicated, but let's break it down:

```
module bin_to_dec #(
    DIGITS = 1 : DIGITS > 0,
    LEADING_ZEROS = 0 : LEADING_ZEROS == 0 || LEADING_ZEROS == 1
  )(
    // minimum number of bits for DIGITS
    input value[$clog2($pow(10, DIGITS))],
    output digits[DIGITS][4]                          // decimal output
  ) {

  var i, j, scale;
  sig remainder[value.WIDTH];                         // running remainder
  // temporary subtraction value
  sig sub_value[value.WIDTH];
  sig blank;                                          // flag for leading zeros

  always {
    for (i = 0; i < DIGITS; i++)                      // for all digits
      // default to invalid number
      digits[i] = d11;

    remainder = value;                                // initialize remainder
    blank = !LEADING_ZEROS;                           // set blank zero flag

    if (value < $pow(10, DIGITS)) {                   // if can be displayed
      for (j = DIGITS - 1; j >= $signed(0); j--) {    // for each digit
        // get the scale for the digit
        scale = $pow(10, j);

        if (remainder < scale) {                      // if this digit is 0
          if (j != 0 && blank)                        // use 10 for blank
            digits[j] = 10;
```

```
            else                                        // or 0 for zero
              digits[j] = 0;
          } else {                                      // digit is 1-9
            blank = 0;                                  // don't blank future zeros
            // default to no subtraction
            sub_value = 0;
            for (i = 9; i >= 1; i--) {                  // for each possible value
              // if remainder is less than value
              if (remainder < (i+1) * scale) {
                digits[j] = i;                          // set digit to this value
                sub_value = i * scale;                  // set subtraction value
              }
            }
            remainder = remainder - sub_value;          // subtract off last digit
          }
        }
      }
    }
  }
```

The two parameters should be fairly straightforward at this point. `DIGITS` specifies the number of digits the decimal number can have, and `LEADING_ZEROS` specifies that 0 or 10 should be used for leading zeros. Basically, since 10 isn't a decimal digit, we can use this as a blank digit value. For a number such as 15 with `LEADING_ZEROS` set to 1 and `DIGITS` set to 4, we would get 0015, but with `LEADING_ZEROS` set to 0, we get 15.

The size of the input `value` is a bit complicated. The function `$pow(a,b)` returns a^b. So here we are using it to get the maximum value of a decimal number with `DIGITS` digits can have, plus 1. For example, if `DIGITS` is 2, the maximum value is 99, and 10^2 is 100. The `$clog2()` function is then used to tell us the minimum number of bits needed to store the maximum value plus 1, minus 1—so really just the minimum bits needed to store the maximum value.

Inside an `always` block, the types `sig` and `var` can be read and written, and their value can change throughout the block. It is best to think of `sig` as a wire. However, when you start using them in `for` loops, they will actually be multiple wires that happen to use the same name. Take a look at Figure 5-3 for one way this module could be implemented.

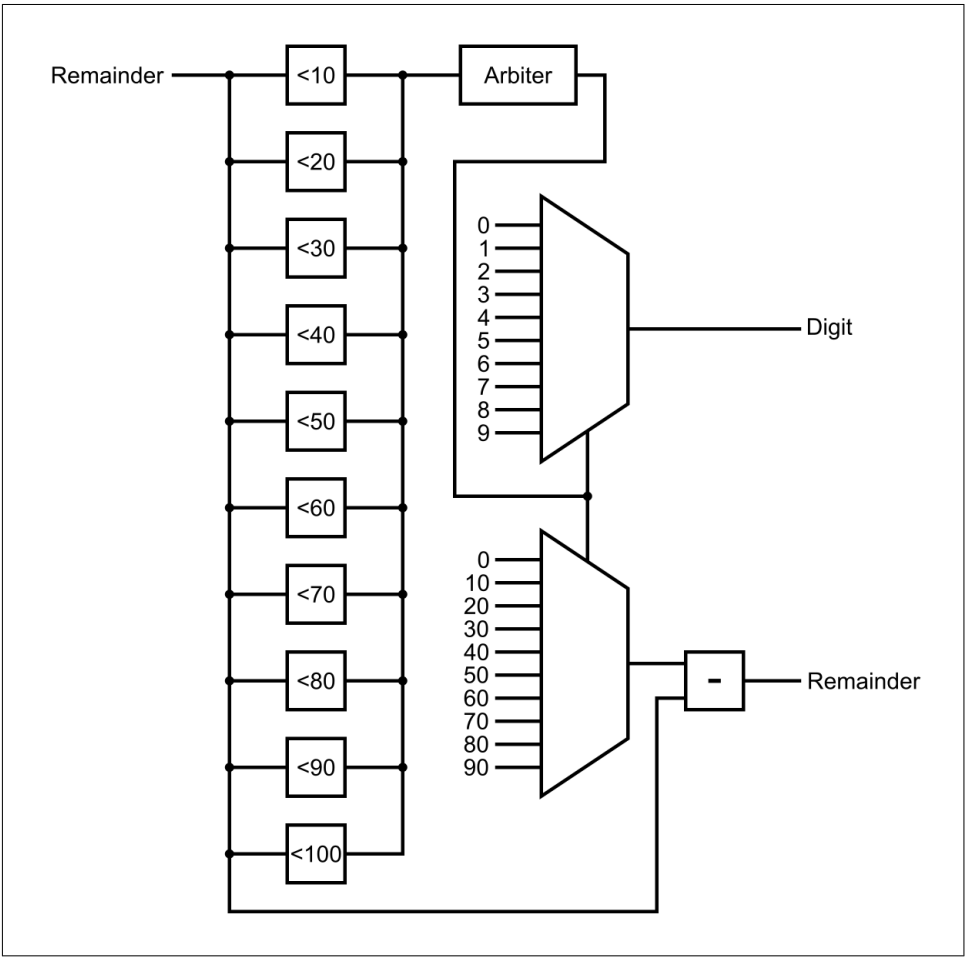

Figure 5-3. Simplified binary-to-decimal schematic

This schematic leaves out the leading zero and overflow detection parts of the design but still illustrates how you could make this module. The part shown would be the part responsible for detecting the 10s place digit. Each digit would have a copy of this circuit, except the constants would be different. Instead of multiples of 10, they would be multiples of that digit's power: 1,000, 100, 10, 1, and so forth.

The first part of the schematic consists of a bank of comparators that detect whether the input is less than all the possible digits (multiplied by the power of this section). These are then fed into an arbiter that will select the smallest number that is still larger than the input. For example, if the input is 24, the arbiter would select the < 30 comparator.

The output from the arbiter is then used to select the digit via a multiplexer being used as a lookup table.

We also need to output the remainder for the next digit, so we then have to subtract off the current digit. Which value we use is selected using the output of the arbiter again but with another multiplexer. If we have 24 as the input, then the 20 would be selected and we would subtract that from the input. The remainder would be 4 and that would get passed to the next stage.

The nice part about this design is that it doesn't use any division. It uses comparisons and subtraction, which are relatively cheap when compared to division.

Note that this is only one way you could realize this design. The tools may take a completely different route that happens to be more efficient. It is, however, important to try to visualize how a circuit you are designing could be realized. This will help you design efficient circuits that are easily realized in hardware.

With an understanding of this component, let's put it to use. Instantiate it in `mojo_top`:

```
    .clk(clk) {
      // The reset conditioner is used to synchronize the reset signal to the FPGA
      // clock. This ensures the entire FPGA comes out of reset at the same time.
      reset_conditioner reset_cond;

      .rst(rst) {
        multi_seven_seg seg_display;
        counter ctr (#TOP(9999), #DIV(22), #SIZE(14));
      }
    }

    bin_to_dec digits (#DIGITS(4));
```

Also note the counter is back and the parameters changed so that it will count from 0–9999.

Then connect it all together in the `always` block:

```
      digits.value = ctr.value;
      seg_display.values = digits.digits;
      seg_display.decimal = 4b0000;

      io_seg = ~seg_display.seg;
      io_sel = ~seg_display.sel;
```

Build and load the project, and the LED displays should show a counter.

The Finite-State Machine

To get our feet wet with FSMs, we will create a stopwatch. This project will build off the project in the previous example, so make sure you have that ready. Basically, we need it to be able to display the stopwatch value on the seven-segment display.

Lucid has a type specifically for FSMs, the `fsm` type. This type is similar to the `dff` type and in fact will instantiate DFFs. The difference is that instead of holding arbitrary binary values, the `fsm` type will hold one of many state constants. For our stopwatch example, we might want the following states: `IDLE`, `RUNNING`, `PAUSED`. To declare an FSM with these states, we could use the following:

```
.clk(clk), .rst(rst) {
  fsm state = {IDLE, RUNNING, PAUSED};
}
```

When reset, the FSM will reset to the first state in the list—in this case, `IDLE`. If for some reason you want to reset to a different state, you can use the parameter `INIT`:

```
fsm state(#INIT(RUNNING)) = {IDLE, RUNNING, PAUSED};
```

FSMs have the same `d` and `q` ports that DFFs have and they act exactly the same. The `q` output is the current state, and the `d` input is the state to transition to. If you don't assign `d` a state, the state will stay the same.

The Stopwatch

Although we could just use a counter as before, it would be much more useful if our stopwatch counted actual seconds. To do this, we need two counters: one that generates a tick at a regular interval, say every tenth of a second, and another that counts the ticks. So how do we generate a tick every tenth of a second?

The clock on the Mojo is 50 MHz, meaning we have 50,000,000 cycles per second. We need to generate a tick every 5,000,000 cycles for a tenth of a second. That means we need a counter that will count from 0 to 4,999,999, which takes exactly 5 million cycles.

To count this high, we need the ceiling of log base 2 of 5,000,000, or 23 bits. If we let the counter overflow naturally, it would count way too long, so we need to reset it whenever its value gets to 4,999,999. Each time we reset it, we can count this as a tick and increment the stopwatch counter.

We can use these two counters with an FSM to control when they are running to create the stopwatch module:

```
module stopwatch (
    input clk,        // clock
    input rst,        // reset
```

```
    input start_stop, // start/stop signal
    output value[14]  // counter value
  ) {

  .clk(clk){
    .rst(rst){
      fsm state = {IDLE, RUNNING, PAUSED};
    }
    dff tenth_ctr[23]; // need to store up to 4,999,999
    dff ctr[14];       // need to store up to 9999
  }

  always {
    value = ctr.q;                      // output counter

    case (state.q) {                    // FSM case statement
      state.IDLE:                       // IDLE: not counting
        tenth_ctr.d = 0;                // reset tenths counter
        ctr.d = 0;                      // reset main counter
        if (start_stop)                 // if start_stop pressed
          state.d = state.RUNNING;      // switch to RUNNING state

      state.RUNNING:                    // RUNNING: increment counters
        tenth_ctr.d = tenth_ctr.q + 1;  // increment tenths counter

        if (tenth_ctr.q == 4999999) {   // if max value
          tenth_ctr.d = 0;              // reset to 0
          ctr.d = ctr.q + 1;            // increment main counter
          if (ctr.q == 9999)            // if max value
            ctr.d = 0;                  // reset to 0
        }

        if (start_stop)                 // if start_stop pressed
          state.d = state.PAUSED;       // switch to PAUSED state

      state.PAUSED:                     // PAUSED: maintain count but do nothing
        if (start_stop)                 // if start_stop pressed
          state.d = state.RUNNING;      // switch back to RUNNING
    }
  }
}
```

FSMs are generally used with `case` statements, as you want to do different things for each state. To access the different state constants, you need to prefix the state name with the FSM's name and a period to separate them. For example, the `IDLE` state can be accessed with `state.IDLE`. This allows you to use the same state names for multiple FSMs if you want.

It is often helpful to draw a state diagram to understand the different states an FSM can be in and under what conditions it will switch between them. Figure 5-4 is the state diagram for the stopwatch.

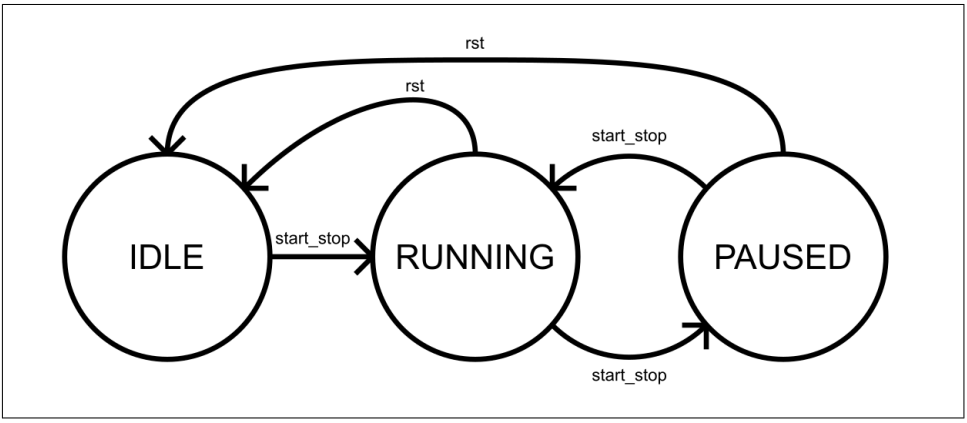

Figure 5-4. Stopwatch state diagram

Basically, the `start_stop` signal will get the FSM to transition from `IDLE` to `RUNNING`, and then it will toggle between `RUNNING` and `PAUSED`. The `rst` signal will always force the FSM back to `IDLE` regardless of the state it is currently in. Note that we don't need to explicitly set `state` to `IDLE` when `rst` is 1 because we hooked `rst` to the reset input of the FSM, and `IDLE` is the first state in the list.

We now need to hook this up in `mojo_top`. We need two buttons, one for the `start_stop` signal and one for reset. We don't really need a separate reset signal because the button on the Mojo acts as the reset by default, but we can use one of the buttons on the IO Shield to reset just the stopwatch.

Because we are using buttons, we need to condition them as before. We can use the Button Conditioner component for this. We also want the `start_stop` signal to be high for only one clock cycle when the start/stop button is pressed so that the FSM changes state only once. To do this, we can connect the Edge Detector component to the output of the Button Conditioner to detect the rising edge.

First we need to instantiate everything:

```
.clk(clk) {
  // The reset conditioner is used to synchronize the reset signal to the FPGA
  // clock. This ensures the entire FPGA comes out of reset at the same time.
  reset_conditioner reset_cond;

  button_conditioner start_stop_button;
  button_conditioner reset_button;
  stopwatch stopwatch;

  edge_detector start_stop_edge (#RISE(1), #FALL(0));

  .rst(rst) {
    multi_seven_seg seg_display;
```

```
    }
  }
```

Notice that we didn't put the `stopwatch` instance in the reset port connection block. This is because we are going to manually connect this later in the `always` block:

```
    start_stop_button.in = io_button[1];        // center button
    reset_button.in = io_button[3];             // left button

    // connect button to edge detector
    start_stop_edge.in = start_stop_button.out;

    stopwatch.start_stop = start_stop_edge.out; // rising edge of start/stop
    stopwatch.rst = rst | reset_button.out;     // reset on rst or reset_button

    digits.value = stopwatch.value;             // display stopwatch value
    seg_display.values = digits.digits;         // digit values to display
    seg_display.decimal = 4b0010;               // turn on second decimal point

    io_seg = ~seg_display.seg;
    io_sel = ~seg_display.sel;
```

Because we want both the global reset and the left button to reset the stopwatch, we can OR the signals together.

If you build and load your project now, you should see *.0* on the LEDs. If you press the center button, the FPGA should start counting in tenths of a second. Pressing the center button again pauses the counter, and pressing the left button resets it back to 0.

CHAPTER 6
Hello AVR

In this chapter, we will design a project that will send the text "Hello World!" through the microcontroller on the Mojo to the USB port and onto your computer. Doing this in hardware is not a trivial project requiring only a few lines. However, it will demonstrate many useful techniques such as *read-only memory* (ROM) and *finite-state machines* (FSMs). ROMs provides a simple way to store a large number of constants, and FSMs allow you to control tasks that require sequential steps. We will also dive into using block *random access memory* (RAM) to save your name as you type it and send it back in a personalized greeting.

The AVR Interface

In this example, we are going to make a new project based on the *AVR Interface* example project because it has the interface to the AVR already set up for us.

Create a new project based on the AVR Interface example project and open *mojo_top.luc*. The starter code for this is similar to before except it already has the `avr_interface` component instantiated and hooked up in the `always` block.

The `avr_interface` module deals with talking to the AVR (the microcontroller on the Mojo) and allows you to easily send and receive data through the USB port, as well as read analog values from the analog pins. In this tutorial, we are going to be using only the USB port.

Look in the `always` block and find the following lines:

```
// unused serial port
avr.tx_data = 8hxx;  // don't care
avr.new_tx_data = 0; // no data
```

Here avr.tx_data is the byte to send, and avr.new_tx_data will send that byte when set to 1. Note that the byte will be ignored if avr.tx_busy is 1, but don't worry about this for now.

There are similar signals for receiving data, and we can "hackily" connect them to echo data by editing the preceding lines to look like the following:

```
// echo serial port
avr.tx_data = avr.rx_data;
avr.new_tx_data = avr.new_rx_data;
```

The reason this is hacky is that we are ignoring the avr.tx_busy flag so bytes can be dropped.

Build the project and load it onto your Mojo. After it's loaded, go to Tools → Serial Port Monitor. This opens a little console that you can type in to send letters over the USB port, and you will see whatever is received. Because we are echoing everything we receive, you should see what you type.

Sending and Receiving Data

Two signals are used for receiving data: avr.new_rx_data and avr.rx_data. The flag avr.new_rx_data, when 1, tells you that the data available on avr.rx_data is valid. The signal avr.rx_data is 8 bits wide and is the data being received.

A similar, but opposite, interface exists for sending data. When sending data, you provide the data on avr.tx_data (also 8 bits wide) and signal when the data is valid by setting the flag avr.new_tx_data to 1. The one catch is that it is possible to provide data way faster than it can be sent. To prevent this, the signal avr.tx_busy needs to be checked before settings avr.new_tx_data to 1. Both of these signals should never be 1 at the same time. To send a byte, first check that avr.tx_busy is 0, and then provide the data on avr.tx_data and set avr.new_tx_data to 1.

If avr.tx_busy is 1, you must wait until it is 0 before trying to send more data. It may be 1 indefinitely if you try to send data and the Mojo isn't plugged in over USB, or an application on the PC side isn't reading the data, causing the buffers to fill up.

ROMs

In this section, we will create a module that will listen for the character h to be received and then respond with "Hello World!". We need a way to store the text "Hello World!". This is where read-only memory, or ROM, comes in. A ROM is pretty similar to the LUT except that it can have many more entries. A ROM has a single input, the address, and a single output, the data. You supply the address, and it gives you the data that corresponds to it.

Create a new module named `hello_world_rom` and add the following:

```
module hello_world_rom (
    input address[4],  // ROM address
    output letter[8]   // ROM output
  ) {

  // text is reversed to make 'H' address [0]
  const TEXT = $reverse("Hello World!\r\n");

  always {
    letter = TEXT[address]; // address indexes 8 bit blocks of TEXT
  }
}
```

Strings in Lucid (and other HDLs) are treated essentially as numbers. The rightmost value is the zeroth index. This is the same order as bits in an array or the array builder syntax.

Strings in Lucid are arranged as a 2D array. The first index is the character, and the second index is the bits of that character. A string of *N* characters is an *N* x 8 array. An exception exists if there is only one character: then it is a single-dimensional array of size 8.

We want the first letter we send to be `H`. That means `H` needs to be at address 0. We could write the `TEXT` constant as `\r\n!dlroW olleH` to accomplish this, but Lucid has the `$reverse()` function that reverses the indices of the outermost dimension of an array. This function can be used only on constants. By using this, we can type the text in a more natural way.

The `always` block section is simple. It outputs the letter of `TEXT` indexed by the `address` input. Notice that `address` is 4 bits wide, as we need it to be able to index 14 characters.

The Greeter

We now need to make a module that will wait for the `h` to be received and then send out the `Hello World!` message. Create a new module called `greeter` for this purpose:

```
module greeter (
    input clk,         // clock
    input rst,         // reset
    input new_rx,      // new RX flag
    input rx_data[8],  // RX data
    output new_tx,     // new TX flag
    output tx_data[8], // TX data
```

```
    input tx_busy      // TX is busy flag
  ) {

  const NUM_LETTERS = 14;

  .clk(clk) {
    .rst(rst) {
      fsm state = {IDLE, GREET};
    }
    dff count[$clog2(NUM_LETTERS)]; // min bits to store NUM_LETTERS - 1
  }

  hello_world_rom rom;

  always {
    rom.address = count.q;
    tx_data = rom.letter;

    new_tx = 0; // default to 0

    case (state.q) {
      state.IDLE:
        count.d = 0;
        if (new_rx && rx_data == "h")
          state.d = state.GREET;

      state.GREET:
        if (!tx_busy) {
          count.d = count.q + 1;
          new_tx = 1;
          if (count.q == NUM_LETTERS - 1)
            state.d = state.IDLE;
        }
    }
  }
}
```

We can use an FSM to switch between waiting for the `h` and sending the message. In the `GREET` state, we can use a counter to keep track of which character we are on and switch back to `IDLE` after we have sent them all.

In the `IDLE` state, to check for an `h` we have to wait for `new_rx` to be 1 and `rx_data` to be `h`. When `new_rx` is 0, the value of `rx_data` should be considered invalid.

Notice that in the `GREET` state, we increase the counter and send a new character only when `tx_busy` is 0. If we didn't have that condition, tha FPGA would try to send all the characters back to back, which would likely get only the first one through.

We can now add the greeter to `mojo_top` and try it out:

```
  .clk(clk) {
    // The reset conditioner is used to synchronize the reset signal to the FPGA
```

```
// clock. This ensures the entire FPGA comes out of reset at the same time.
reset_conditioner reset_cond;

.rst(rst){
  // the avr_interface module is used to talk to the AVR
  // for access to the USB port and analog pins
  avr_interface avr;

  greeter greeter; // instance of our greeter
}
```

We also need to connect it up in the `always` block:

```
// connect the greeter
greeter.new_rx = avr.new_rx_data;
greeter.rx_data = avr.rx_data;
avr.new_tx_data = greeter.new_tx;
avr.tx_data = greeter.tx_data;
greeter.tx_busy = avr.tx_busy;
```

Note that these will replace the lines we had for echoing data back.

Build and load the project onto your Mojo. Then go to Tools → Serial Port Monitor to open the Serial Port Monitor. This allows you to send and receive text. Make sure the correct serial port is selected in the drop-down. If you just programmed your Mojo, the port should already be selected. Then try typing **h** into the monitor. Anything you type will be sent to the Mojo, and anything the Mojo sends back will be displayed. Each `h` you send should result in "Hello World!" showing up.

Getting Personal with RAM

The last example was good to get your feet wet, but it was so impersonal. It would be much better if we could greet you by name. In this example we are going to ask you for your name and then respond by greeting you personally. Create a new project based on the AVR Interface example project.

As the title of this section alludes to, we are going to use RAM in this example. There are a few types of RAM, and we are going to use the most basic one, a simple single-port RAM. There is a component for this in the Components Library. It's called *Simple RAM* and is under the Memory category.

Although you *could* code a RAM yourself, if it doesn't follow a specific template, the tools won't recognize it as RAM and will fail to properly utilize the block RAM that exists in the FPGA. FPGAs have something known as the *general fabric*, which is flexible and where most of your design gets implemented, but they also have special resources to help with certain design elements such as RAM, ROM (RAM can be initialized so it can act like a ROM), multiplication, and special I/O functions. By using

the Simple RAM component, you know the tools will recognize it is RAM and will use the FPGA's resources efficiently.

However, in order to fit the specific template, the `simple_ram` module is coded in Verilog. You can take a look at the entire *simple_ram.v* file, which is now in the Components branch of your project tree, but we will just look at the module declaration:

```
module simple_ram #(
    parameter SIZE = 1,  // size of each entry
    parameter DEPTH = 1  // number of entries
  )(
    input clk,                          // clock
    input [$clog2(DEPTH)-1:0] address, // address to read or write
    output reg [SIZE-1:0] read_data,   // data read
    input [SIZE-1:0] write_data,       // data to write
    input write_en                      // write enable (1 = write)
  );
```

Mixing Lucid and Verilog

The Mojo IDE has limited support for mixing Lucid and Verilog in your project. When using Verilog, many of the IDE's convenient features, such as real-time error checking, are limited.

Just like Lucid, Verilog has parameters that act the same way. However, there isn't a way to specify constraints on them, and they require a default value.

The inputs and outputs are similar as well, but the array sizing is a bit different. First, the array size comes before the array's name. It also takes the form `[End Index : Start Index]` instead of simply `[Size]`. The start index should basically always be 0, so the size is the end index plus 1. That plus 1 is why most of the end indices have a minus 1. For example, `write_data` has a size of `SIZE`.

This RAM works similarly to the ROM from before, with a few differences. The ROM didn't use a clock (which really made it more like a LUT), and the output was available on the same clock cycle as we specified in an address. When reading from the RAM, the data will be available a clock cycle after the address is specified.

To write to the RAM, you supply an address and data on `address` and `write_data`, respectively, and set `write_en` to 1. The RAM is always reading, so specifying an address will cause that data to show up on `read_data` in one clock cycle.

We can use the RAM to make a new greeter module. You can find the full source at "Greeter Module" on page 205.

With this greeter, we didn't bother putting the text into a separate module, as it is just as easy to access directly as an array. In HELLO_TEXT, we are using the @ character to signify where to put the name:

```
// reverse so index 0 is the left most letter
const HELLO_TEXT = $reverse("\r\nHello @!\r\n");
const PROMPT_TEXT = $reverse("Please type your name: ");
```

Even though the simple RAM module is written in Verilog, we can use it just like any other module:

```
simple_ram ram (#SIZE(8), #DEPTH($pow(2,name_count.WIDTH)));
```

We set SIZE to 8, which is how big each entry is, and DEPTH to 2 to the power of our name counter, which is the maximum number of letters we'll allow a name to have. The DEPTH parameter specifies the number of entries in the RAM. It is nice if this is a power of 2 so that there are no invalid addresses, but it doesn't have to be.

In the IDLE state, we reset all the counters and wait for anything to be sent over the serial port so we know someone is there. We then transition to PROMPT to ask the user for his or her name:

```
// IDLE: Reset everything and wait for a new byte.
state.IDLE:
  hello_count.d = 0;
  prompt_count.d = 0;
  name_count.d = 0;
  if (new_rx) // wait for any letter
    state.d = state.PROMPT;
```

The PROMPT state is basically the same as the GREET state in the old greeter. We pump out all the letters in PROMPT_TEXT as fast as we can and then change to the LISTEN state:

```
// PROMPT: Print out name prompt.
state.PROMPT:
  if (!tx_busy) {
    prompt_count.d = prompt_count.q + 1;   // move to the next letter
    new_tx = 1;                            // send data
    tx_data = PROMPT_TEXT[prompt_count.q]; // send letter from PROMPT_TEXT
    if (prompt_count.q == PROMPT_TEXT.WIDTH[0] - 1) // no more letters
      state.d = state.LISTEN;              // change states
  }
```

In the LISTEN state, we wait for data to be received and save each character to the RAM, incrementing the address each time. If we run out of space, or receive a newline character, we switch to the HELLO state:

```
// LISTEN: Listen to the user as they type his/her name.
state.LISTEN:
  if (new_rx) { // wait for a new byte
    ram.write_data = rx_data;        // write the received letter to RAM
    ram.write_en = 1;                // signal we want to write
    name_count.d = name_count.q + 1; // increment the address

    // We aren't checking tx_busy here that means if someone types super
    // fast we could drop bytes. In practice this doesn't happen.

    // only echo non-new line characters
    new_tx = rx_data != "\n" && rx_data != "\r";
    tx_data = rx_data; // echo text back so you can see what you type

    // if we run out of space or they pressed enter
    if (&name_count.q || rx_data == "\n" || rx_data == "\r") {
      state.d = state.HELLO;
      name_count.d = 0; // reset name_count
    }
  }
```

The snippet `&name_count.q` is a useful technique for checking whether something is maxed out. The `&` operator there is the AND reduction operator, which will AND all the bits of `name_count.q` together, effectively testing whether they are all 1.

Finally, the `HELLO` state prints the `HELLO_TEXT` but with the `@` character replaced with the name stored in the RAM. It does this by printing `HELLO_TEXT` normally, but when the `@` is reached, it switches to printing from the RAM. After the name from RAM has been printed, the counter indexing `HELLO_TEXT` is incremented to bypass the `@` and it resumes normal printing until it reaches the end. It then cycles back to the `IDLE` state to greet you again:

```
// HELLO: Prints the hello text with the given name inserted
state.HELLO:
  if (!tx_busy) { // wait for tx to not be busy
    if (HELLO_TEXT[hello_count.q] != "@") { // if we are not at the sentry
      hello_count.d = hello_count.q + 1;    // increment to next letter
      new_tx = 1;                           // new data to send
      tx_data = HELLO_TEXT[hello_count.q];  // send the letter
    } else {                                // we are at the sentry
      name_count.d = name_count.q + 1;      // increment name_count letter

      // if we are not at the end
      if (ram.read_data != "\n" && ram.read_data != "\r")
        new_tx = 1;                                     // send data

      tx_data = ram.read_data;              // send the letter from the RAM

      // if we are at the end of the name or out of letters to send
      if (ram.read_data == "\n" || ram.read_data == "\r" || &name_count.q) {
        // increment hello_count to pass the sentry
```

```
      hello_count.d = hello_count.q + 1;
    }
  }

  // if we have sent all of HELLO_TEXT
  if (hello_count.q == HELLO_TEXT.WIDTH[0] - 1)
     state.d = state.IDLE; // return to IDLE
}
```

CHAPTER 7
Mixing Colors with an RGB LED

In this chapter, we will hook up a red, green, and blue (RGB) LED to the Mojo and use *pulse-width modulation* (PWM), to fade the colors in and out. We will then explore how to use the Register Interface to control the LED. The Register Interface provides a simple way to interface your designs with your computer without having to parse streams of characters in the FPGA. We will be using this to allow you to set the color of the LED from your computer.

To complete these examples, you need an RGB LED and three resistors with a value close to 330 ohms. The LED I'm using is a common anode type (LEDs share a single positive pin, and the negative sides are used to control them individually), but a common cathode LED will work as well. You may also want a breadboard and some jumper wires.

We are going to use the pins 40, 50, and 51 on the Mojo. They are located in the top-left corner and are colored red in Figure 7-1.

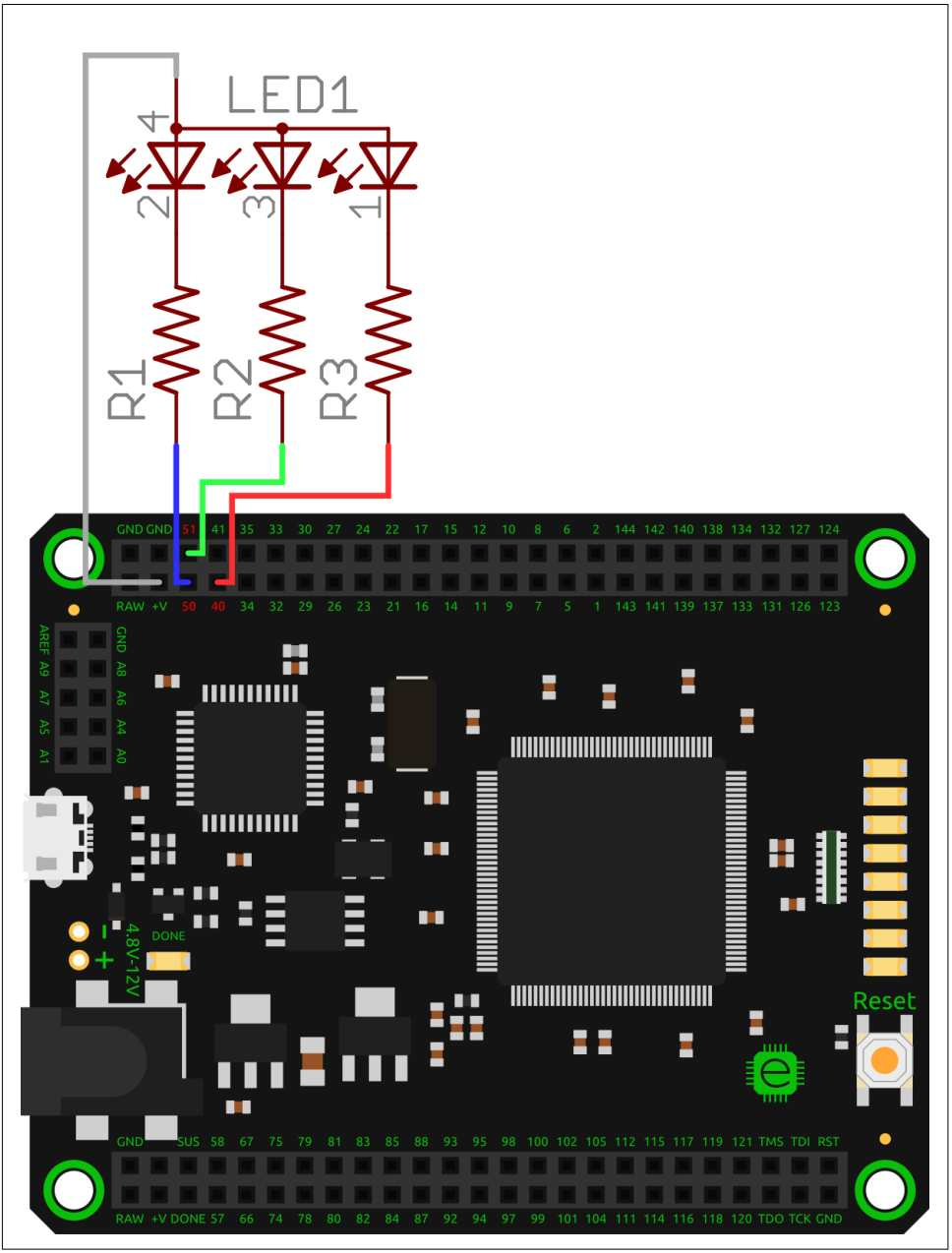

Figure 7-1. Mojo pinout with pins 40, 50, and 51 highlighted in red

Take your LED and connect the three LED color channels to the three pins. It doesn't matter which pins you choose, as you'll see later, but I used 40 for red, 51 for green,

and 50 for blue. All that is important is that you know which is which. Make sure that you connect the LED with the resistors between it and the Mojo. If you have a common anode LED, connect the common pin to V+. If you have a common cathode LED, connect the common pin to GND.

External I/O Constraints

Create a new project based on the Base Project example.

We need to create three new outputs in `mojo_top` that will be the outputs for the LED. The first step is to add them to the module declaration:

```
module mojo_top (
    input clk,              // 50MHz clock
    input rst_n,            // reset button (active low)
    output led [8],         // 8 user controllable LEDs
    input cclk,             // configuration clock, AVR ready when high
    output spi_miso,        // AVR SPI MISO
    input spi_ss,           // AVR SPI Slave Select
    input spi_mosi,         // AVR SPI MOSI
    input spi_sck,          // AVR SPI Clock
    output spi_channel [4], // AVR general purpose pins
    input avr_tx,           // AVR TX (FPGA RX)
    output avr_rx,          // AVR RX (FPGA TX)
    input avr_rx_busy,      // AVR RX buffer full
    output red,             // LED red channel
    output green,           // LED green channel
    output blue             // LED blue channel
  ) {
```

Don't forget to add a comma after `avr_rx_busy` when adding the new outputs.

We now need to tell the tools what I/O pins to use for these outputs. If we don't, the tools will pick some unused pins that it finds convenient. This can be helpful if you are working on an FPGA design before laying out your actual board, but you'll almost always want to constrain the I/O location to a specific pin.

This is where *user constraint files* (UCFs) come into play. These are Xilinx-specific files that tell the tools various things about your design. They can be used to give you a ton of control over how your design is laid out inside the FPGA, but we will be using them only to constrain which I/O pins are used. If you want to look into the details of UCFs, see the official documentation.

Take a look at the *mojo.ucf* file under the Constraints branch in the project tree:

```
NET "clk" TNM_NET = clk;
TIMESPEC TS_clk = PERIOD "clk" 50 MHz HIGH 50%;

NET "clk" LOC = P56 | IOSTANDARD = LVTTL;
NET "rst_n" LOC = P38 | IOSTANDARD = LVTTL;
```

```
NET "cclk" LOC = P70 | IOSTANDARD = LVTTL;

NET "led<0>" LOC = P134 | IOSTANDARD = LVTTL;
NET "led<1>" LOC = P133 | IOSTANDARD = LVTTL;
NET "led<2>" LOC = P132 | IOSTANDARD = LVTTL;
NET "led<3>" LOC = P131 | IOSTANDARD = LVTTL;
NET "led<4>" LOC = P127 | IOSTANDARD = LVTTL;
NET "led<5>" LOC = P126 | IOSTANDARD = LVTTL;
NET "led<6>" LOC = P124 | IOSTANDARD = LVTTL;
NET "led<7>" LOC = P123 | IOSTANDARD = LVTTL;

NET "spi_mosi" LOC = P44 | IOSTANDARD = LVTTL;
NET "spi_miso" LOC = P45 | IOSTANDARD = LVTTL;
NET "spi_ss" LOC = P48 | IOSTANDARD = LVTTL;
NET "spi_sck" LOC = P43 | IOSTANDARD = LVTTL;
NET "spi_channel<0>" LOC = P46 | IOSTANDARD = LVTTL;
NET "spi_channel<1>" LOC = P61 | IOSTANDARD = LVTTL;
NET "spi_channel<2>" LOC = P62 | IOSTANDARD = LVTTL;
NET "spi_channel<3>" LOC = P65 | IOSTANDARD = LVTTL;

NET "avr_tx" LOC = P55 | IOSTANDARD = LVTTL;
NET "avr_rx" LOC = P59 | IOSTANDARD = LVTTL;
NET "avr_rx_busy" LOC = P39 | IOSTANDARD = LVTTL;
```

The first two lines tell the tools that the `clk` input is a clock with a frequency of 50 MHz and a duty cycle of 50% (high half the time). The rest of the lines are the pins to use for the inputs and outputs in the Base Project.

Although you could typically just add some extra pin location constraints to the same file, the *mojo.ucf* file is a shared file and can't be edited. Instead, we need to create a new UCF to add our constraints to. Click the Add File icon and create a new file called *led.ucf*. Make sure to select User Constraints as the file type.

In the new *led.ucf* file, add the following:

```
NET "red" LOC = P40 | IOSTANDARD = LVTTL;
NET "green" LOC = P51 | IOSTANDARD = LVTTL;
NET "blue" LOC = P50 | IOSTANDARD = LVTTL;
```

If you didn't use the same pins as me, you can change the numbers for each one to match your setup. All the pin numbers are labeled on the board, and their names are simply the number prefixed with a `P`.

Also note that for all the pin location constraints, we also specify the `IOSTANDARD` constraint. This is required, as the default I/O standard is `LVCMOS25`, which requires 2.5 V I/O, and the Mojo uses 3.3 V. If you fail to specify this, or specify a different one, the tools will complain and your project will fail to build.

If your I/O pin is an array, you can use `<BIT>` appended to the net name to specify the location of each bit. See the `led` pin locations in *mojo.ucf* for an example.

With the pin definitions now in place, we can test out the LED. In the `always` block in `mojo_top`, add the following:

```
red = 0;
green = 1;
blue = 1;
```

This is for a common anode LED that will turn on when the output is 0. Invert the values if you are using a common cathode LED. If you build and load your project now, the RGB LED should be red.

Getting Fancy with PWM

Turning on different colors is great and all, but RGB LEDs need to have their colors blended. We can use a technique known as *pulse-width modulation* (PWM) to create the illusion of varying the brightness of the colors. PWM works by turning the LED on and off really fast. The percent of the time the LED is on is known as the *duty cycle*. A 25% duty cycle means the LED is on for a quarter of the time. The duty cycle is independent of the frequency.

We don't particularly care about the frequency for the LED, as long as it isn't really high or too low. We just need to make sure it is high enough that we can't see a flicker. If you go too high, you start losing noticeable power to capacitance in the LED and board.

To generate the pulse train, we can use a continuously running counter. To set the duty cycle, we can compare the value of the counter to a value. If the counter is less than that value, we output 1; otherwise we output 0. The duty cycle we generate is then proportional to the comparison value divided by the maximum counter value. If we set the comparison value to half of the counter's max value, we will get a 50% duty cycle. Figures 7-2 and 7-3 show how the duty cycle changes when we change the compare value.

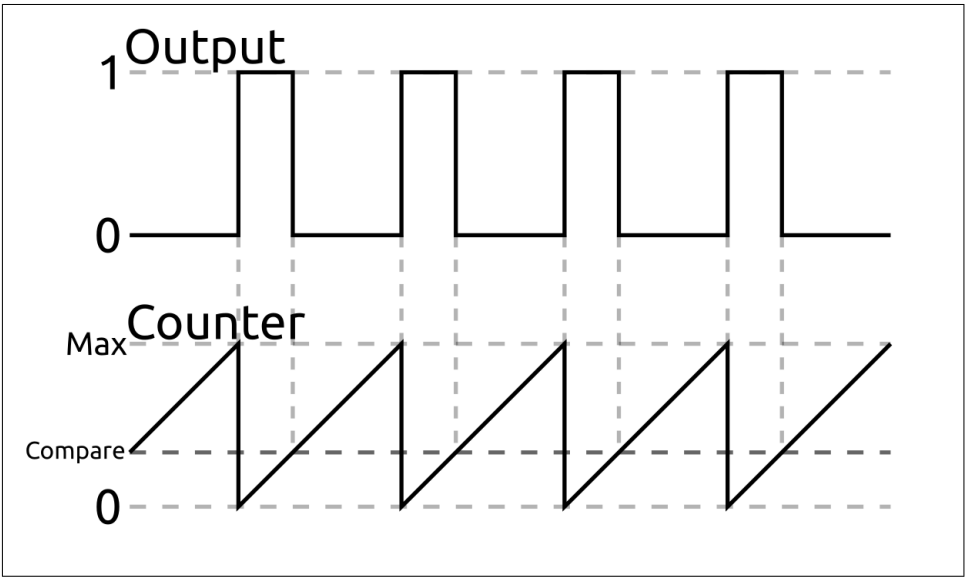

Figure 7-2. PWM based off a counter with a 33% duty cycle

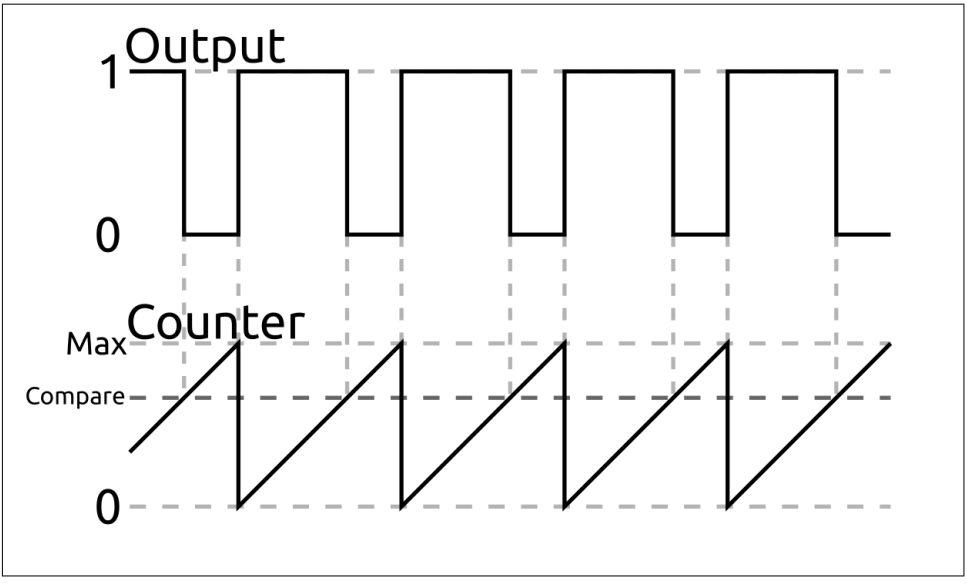

Figure 7-3. PWM based off a counter with a 66% duty cycle

As you may have guessed, there is a component for creating a PWM signal. Add the Pulse Width Modulator component to your project. It can be found in the LED Effects category.

This component is pretty simple, but it has a few extras that make it produce a glitch-free PWM signal. A glitch in the PWM signal can happen if you change the compare value mid cycle. For example, let's assume an 8-bit counter, so it has a range of 0–255. If the compare value is 25, and the counter value is 120, the pulse will already have been sent, and the output will be low. If we then change the compare value to 130, the output will be high for 10 counts. This extra pulse will be shorter than anything we wanted and will screw up the frequency. Although this doesn't really matter for the LED, since you won't notice it with your eyes, it matters if you use the PWM module to control a servo or anything that measures the pulse widths.

To fix this, we update the compare value only when the counter is 0. The output will always be 1 at this time, so there is no chance of causing a glitch. This means we need to save the current compare value as well as the value to update it with and indicate whether it needs to be updated:

```
module pwm #(
    WIDTH = 8 : WIDTH > 0, // resolution of PWM counter
    TOP = 0   : TOP >= 0,  // max value of counter
    DIV = 0   : DIV >= 0   // clock pre-scaler
  )(
    input clk,           // clock
    input rst,           // reset
    input value[WIDTH], // duty cycle value
    input update,        // new value flag
    output pulse         // PWM output
  ){

  .clk(clk) {
    .rst(rst) {
      counter ctr(#SIZE(WIDTH), #DIV(DIV), #TOP(TOP));
      dff curValue[WIDTH];
      dff needUpdate;
    }
    // nextValue doesn't need reset
    dff nextValue[WIDTH];
  }

  always {
    // if ctr.value is 0 and we need to update
    if (!|ctr.value && needUpdate.q) {
      curValue.d = nextValue.q; // set our new value
      needUpdate.d = 0;         // don't need update now
    }

    // if value is valid
    if (update) {
      nextValue.d = value; // save it
      needUpdate.d = 1;    // flag for update
    }
```

```
    // if the counter is less than the set
    // value output 1, otherwise output 0
    pulse = ctr.value < curValue.q;
  }
}
```

We can use three of these to control the colors of the LED in `mojo_top`:

```
.clk(clk) {
  // The reset conditioner is used to synchronize the reset signal to the FPGA
  // clock. This ensures the entire FPGA comes out of reset at the same time.
  reset_conditioner reset_cond;

  .rst(rst) {
    pwm r;
    pwm g;
    pwm b;
  }
}
```

We can then connect them to the LED colors and assign some values:

```
  red = ~r.pulse;
  green = ~g.pulse;
  blue = ~b.pulse;

  r.update = 1;
  g.update = 1;
  b.update = 1;

  r.value = 8d128;
  g.value = 8d10;
  b.value = 8d230;
```

Because the LED I'm using is a common anode, I needed to invert the pulse signal. Directly connect the pulse signal if you are using a common cathode LED.

You can play with the values you assign to the colors to get different mixes. If you build and load the preceding values, you should get a nice purplish color.

Let's get a little fancier and make the color dynamic. It's always fun to make RGB LEDs fade through the rainbow, as shown in Figure 7-4. To do this, we need to cycle the three channels in a specific way.

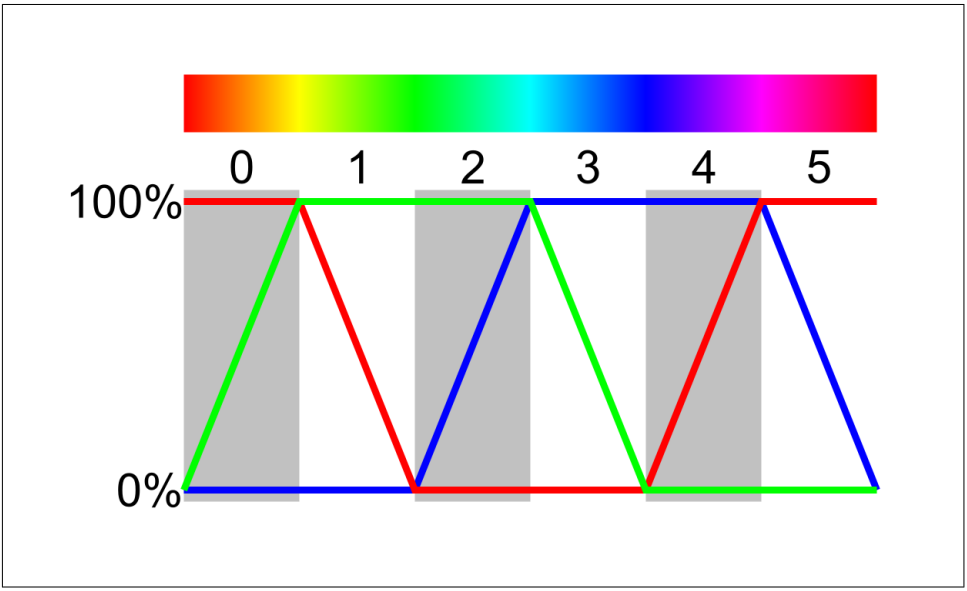

Figure 7-4. RGB fade pattern

Basically, we always want one channel to be full on, one to be off, and one to be turning on or off. Each color takes turns being the one to turn on or off, and we can break the pattern into six regions of change.

We need a way to generate the transitional values as well as figure out which region we are in. This is easily done by using a counter. We can use an 11-bit counter with the 8 LSBs being used for the transitional values and the 3 MSBs being used to keep track of the region. You can think of this as two counters, the 8 LSBs being one, and the 3 MSBs being another, that increment each time the first counter overflows. The only catch is that we have six regions, but 3 bits can have eight different values, so we need to set a top value for the counter:

```
.clk(clk) {
  // The reset conditioner is used to synchronize the reset signal to the FPGA
  // clock. This ensures the entire FPGA comes out of reset at the same time.
  reset_conditioner reset_cond;

  .rst(rst) {
    pwm r;
    pwm g;
    pwm b;

    counter ctr (#SIZE(11), #DIV(17), #TOP(b10111111111));
  }
}
```

You will need to add the Counter component to your project.

The value for `TOP` was chosen so that it would rest right at the end of the sixth region, when the 3 MSBs have the value 5 and the 8 LSBs are maxed out. We also set `DIV` so that the LED fades at a reasonable speed.

We can then replace the constant value assignments with a `case` statement:

```
case (ctr.value[10:8]) {
  0:
    r.value = hff;
    g.value = ctr.value[7:0];
    b.value = h00;
  1:
    r.value = ~ctr.value[7:0];
    g.value = hff;
    b.value = h00;
  2:
    r.value = h00;
    g.value = hff;
    b.value = ctr.value[7:0];
  3:
    r.value = h00;
    g.value = ~ctr.value[7:0];
    b.value = hff;
  4:
    r.value = ctr.value[7:0];
    g.value = h00;
    b.value = hff;
  5:
    r.value = hff;
    g.value = h00;
    b.value = ~ctr.value[7:0];
  default:
    r.value = hxx; // don't care, should never be here
    g.value = hxx;
    b.value = hxx;
}
```

Notice we are using the 3 MSBs as the value for the `case` statement. This allows us to set different values for the six regions. We have a `default` case, even though it should never be used so that the three `pwm` values are guaranteed to have a value assigned, even if the value is *don't care*.

To generate the down slopes, we can invert the bits of the counter. This will cause it to count down from 255 to 0. This is code equivalent to doing 255 minus the counter value, but inversion is a lot cheaper than subtraction.

If you build and load the project now, the RGB LED should smoothly fade through the rainbow of colors.

Register Interface

In this section, we are going to use the Register Interface to control the RGB LED from the Mojo IDE. The Register Interface component takes control of the serial port communications with the AVR and breaks out a simple-to-use address-based interface. Basically, it allows the FPGA to act sort of like RAM for your computer. Your computer can issue reads and writes to addresses. The difference is that the FPGA can interpret these reads and writes however you want. We are going to use three addresses to allow your computer to read and write the brightness values for the different color channels of the RGB LED.

Create a new project based on the *Reg Interface* example. This example project has the Register Interface component already wired up for us. This example project makes the LEDs on the Mojo accessible to your computer on address 0. Build and load the project without any modifications to test it out.

With the project loaded, open the Register Interface tool via Tools → Register Interface. This allows you to manually send read/write commands to your designs. Type **0** in for the address, and enter a number between 0 and 255 for the value, as shown in Figure 7-5.

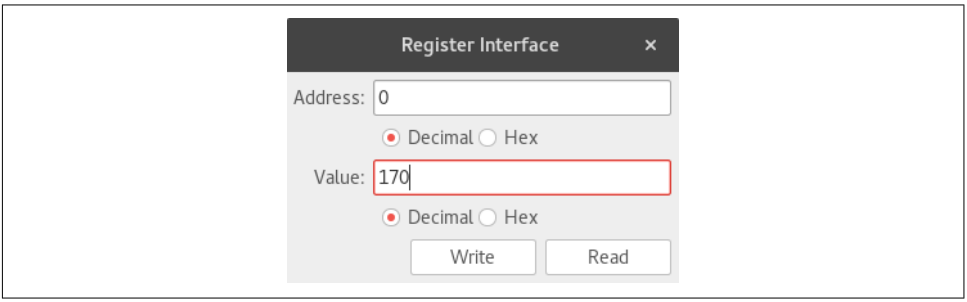

Figure 7-5. Register Interface

The LEDs on your Mojo should turn on, displaying the number you just wrote (in binary, of course). Try out a few numbers. If you change the address to something other than 0, you'll notice that it doesn't do anything for writes and that reads fail. This is because our design ignores reads and writes to other addresses.

Take a look at the *reg_interface.luc* file in the Components branch of the project tree. In the beginning of the file, we have a new type of code block, global definitions:

```
global Register {
  struct master {                          // master device outputs
    new_cmd,                               // 1 = new command
    write,                                 // 1 = write, 0 = read
    address[32],                           // address to read/write
    data[32]                               // data to write (ignore on reads)
```

```
    }
    struct slave {                          // slave device outputs
      data[32],                             // data read from requested address
      drdy                                  // read data valid (1 = valid)
    }
  }
```

The name of the `global` block comes right after the `global` keyword and must start with a capital letter and have at least one lowercase letter. Its content consists of constant and/or struct definitions. A *struct* is similar to an array, but the members are accessed by names instead of numbers. Unlike an array, the members can have different sizes. Structs aren't instantiated directly, but rather one is defined that can then be used to size a different type (`dff`, `fsm`, `sig`, etc.).

In this `global` block, two structs are defined that are used in the port declaration of the `reg_interface` module:

```
// Register Interface
output<Register.master> regOut,             // register outputs
input<Register.slave> regIn                 // register inputs
```

To use a struct, add it to the type with the `<STRUCT_NAME>` syntax. You can also make a struct into an array of structs by using the same array syntax as before.

Notice that the names of the structs are prefixed with the name of their `global` block and separated by a period. This is so that you can have multiple `global` blocks with the same constant or struct names. It also ensures that global definitions won't conflict with any local definitions in your modules.

Let's look at `mojo_top` to see how the structs are used:

```
reg.regIn.drdy = 0;                     // default to not ready
reg.regIn.data = 32bx;                  // don't care

if (reg.regOut.new_cmd) {               // new command
  if (reg.regOut.write) {               // if write
    if (reg.regOut.address == 0) {      // if address is 0
      leds.d = reg.regOut.data[7:0];    // write the LEDs
    }
  } else {                              // if read
    if (reg.regOut.address == 0) {      // if address is 0
      reg.regIn.data = leds.q;          // read the LEDs
      reg.regIn.drdy = 1;               // signal data ready
    }
  }
}

led = leds.q;                           // connect the dff
```

The port is accessed the same way as any other port, with the module instance name followed by a period and the port name. However, because the port is a struct, we can also access the individual struct members with their names.

In this section of code, we wait for a new command. If it is a write, we use the provided value to set the value of the DFF we are using to control the LEDs. If it is a read, we return the value that the LEDs are currently set to. It's important that `regIn.drdy` be 0 until you are responding to a read request. By default, you have about a quarter-second at most to respond to a read request. If you don't respond to the read, a timed-out error will be thrown. In the preceding example, we respond only to reads to the address 0, so if you try to read anything else, you will get a timed-out error.

Let's modify `mojo_top` so that we can control the RGB LED by using the Register Interface. First, we need to add some of the code from the previous example. Create a new UCF and add the lines to connect the three LED pins to the *red*, *green*, and *blue* ports that you need to add as well. Then add the Pulse Width Modulator component to your project and instantiate three of them as before.

We can then modify the preceding code to write to the PWM module instances when addresses 1, 2, or 3 are written to:

```
reg.regIn.drdy = 0;                 // default to not ready
reg.regIn.data = 32bx;              // don't care

r.update = 0;                       // default to no update
g.update = 0;
b.update = 0;
r.value = 8hxx;                     // when update = 0, value doesn't matter
g.value = 8hxx;
b.value = 8hxx;

red = ~r.pulse;                     // connect to LEDs
green = ~g.pulse;
blue = ~b.pulse;

if (reg.regOut.new_cmd) {           // new command
  if (reg.regOut.write) {           // if write
    case (reg.regOut.address) {
      0:
        leds.d = reg.regOut.data[7:0];
      1:
        r.value = reg.regOut.data[7:0];
        r.update = 1;
      2:
        g.value = reg.regOut.data[7:0];
        g.update = 1;
      3:
        b.value = reg.regOut.data[7:0];
        b.update = 1;
    }
```

```
    } else {                            // if read
      if (reg.regOut.address == 0) {    // if address is 0
        reg.regIn.data = leds.q;        // read the LEDs
        reg.regIn.drdy = 1;             // signal data ready
      }
    }
  }

  led = leds.q;                         // connect the dff
```

Because we care about more than one address, we can replace the `if` statement with a `case` statement. The PWM module will change its value only when `update` is 1. We can take advantage of that by setting it to 1 only when a new value is written to the corresponding address. However, we don't have access to the current PWM values, so we can't provide them back on a read. In this case, it's not a big deal, so we still ignore reads to these addresses.

If you build and load the project, you can then use the Register Interface tool to write to addresses 1–3. Writing to address 1 will change the red color, 2 for green, and 3 for blue. Remember, the PWM module uses 8 bits, so you can write a value between 0 and 255, with 255 being full brightness. Try writing 10 to address 1, 191 to address 2, and 10 to address 3. That is Embedded Micro's green.

Hopefully, this gives you an idea of how easy it is to create a fairly complex interface to your FPGA. The Register Interface isn't limited to manual reads and writes from the tool built into the IDE. You can write your own programs that utilize the interface. A basic Eclipse Java project shows you how to talk to it using Java; you can download the project at Embedded Micro.

You don't have to use Java, and you can write your own library to interface in any language that supports reading and writing to a serial port.

Every request starts with 5 bytes being sent. The first byte is the command byte, and the next four are the address (32 bits = 4 bytes) sent with the least significant byte first.

In many cases, you will want to read or write to many consecutive addresses, or perhaps the same address many times. Issuing the entire command each time would be inefficient, so the command byte contains info for consecutive or multiple read/write requests in one.

The MSB (bit 7) of the command byte specifies whether the command is a read (0) or write (1). The next bit (bit 6) specifies whether consecutive addresses should be read/written (1) or whether the same address should be read/written multiple times (0). The 6 LSBs (bits 5–0) represent the number of read/write requests that should be generated. Note that the number of requests will always be 1 more than the value of these bits. That means if you want to read or write a single address, they should be set to 0. Setting them to 1 will generate two read or write requests.

If you send a write request, after sending the command byte and address, you continue sending the data to be written. Data is always sent as 4 bytes per write request, and the least significant byte should be sent first.

For reads, after sending the command byte and address, you wait for the data to be returned. Data is returned in least-significant-byte order. Note that you may not always receive all the data you ask for if there is an issue with the FPGA design (i.e., the requested data is never presented, as in our LED example).

Let's take a look at an example. If you want to write to addresses 5, 6, and 7, you could issue the following request:

```
0xC2 (1100 0010), 0x05, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x02, 0x00,
0x00, 0x00, 0x03, 0x00, 0x00, 0x00
```

This would write 1 to address 5, 2 to address 6, and 3 to address 7. If you changed the command byte to `0x82` (1000 0010), you would write 1 to address 5, 2 to address 5, and then 3 to address 5 (in that order).

Issuing a read follows the exact same format, except the data bytes are received instead of sent. A single command can generate up to 64 read/write requests. If you need to read/write more, you need to issue separate commands.

CHAPTER 8
Analog Inputs

There are eight analog inputs on the Mojo, accessible though the 10-pin connector. However, FPGAs don't usually have any analog circuitry built in. Because of this, these analog inputs are from the microcontroller on the Mojo. After the microcontroller does its job of programming the FPGA, it becomes a USB-to-serial converter as well as an analog-to-digital converter (ADC).

In this tutorial, we are going to read the analog inputs and use their values to vary the brightness of the eight LEDs on the Mojo.

The AVR Interface

We've used the AVR Interface component before, in Chapter 6. However, this time we are going to use the ADC portion of it instead of the serial portion. The ADC on the AVR takes a voltage input in the range of 0–3.3 V and converts it to a number from 0–1,023 (a 10-bit value). As before, we will start by creating a new project based on the *AVR Interface* example project.

If you take a look at the `avr_interface` module, there are four signals we care about when trying to read the analog inputs:

```
// ADC Interface Signals
input channel[4],          // ADC channel to read from, use hF to disable ADC
output new_sample,         // new ADC sample flag
output sample[10],         // ADC sample data
output sample_channel[4],  // channel of the new sample
```

You'll probably notice that only one of these is an input. That means the value we have to worry about providing is `channel`, which is the number of the input we want to read. If you look at your Mojo, there are eight analog inputs, but they are num-

bered a little strangely. The inputs are A0, A1, and A4–A9. A2 and A3 are missing. They simply don't exist for this package of the microcontroller.

To use the ADC, you need to set `channel` to a channel to read. Samples will then start to come in. If you don't need to use the ADC, you should set `channel` to any invalid value, such as 15. When the ADC isn't being used, the microcontroller will have more time to service the USB-to-serial portion of its job, and the bandwidth will be substantially higher. Remember, it isn't an FPGA, so the single processor has to split its time between tasks.

When `channel` is a valid value, new samples will start to show up. When a new sample arrives, `new_sample` will be 1. This tells you that the data on `sample` and `sample_channel` is valid. The signal `sample` is the actual sample value, while `sample_channel` is the channel the sample was taken from. If you set `channel` to a constant, you can safely ignore `sample_channel`, but if you change `sample`, you should verify that `sample_channel` matches the channel you are looking for, as it could be from the old channel.

To test out the ADC, let's first sample only one channel. Add the Pulse Width Modulator component to your project again. Then change line 41 in `mojo_top` to the following:

```
    avr.channel = h0;           // ADC set to channel 0
```

By setting `avr.channel` to 0, we will be continuously reading from *A0*.

We then need to instantiate the PWM module and connect it to the AVR Interface:

```
.clk(clk) {
  // The reset conditioner is used to synchronize the reset signal to the FPGA
  // clock. This ensures the entire FPGA comes out of reset at the same time.
  reset_conditioner reset_cond;

  .rst(rst){
    // the avr_interface module is used to talk to the AVR for access to the
    // USB port and analog pins
    avr_interface avr;

    pwm pwm; // PWM module for the LEDs
  }
}
```

Add this to the end of the `always` block:

```
    pwm.update = avr.new_sample; // update when we have a new sample
    pwm.value = avr.sample[9:2]; // use the 8 MSBs

    led = 8x{pwm.pulse};        // connect PWM output to all the LEDs
```

We update the PWM value only when we have a new sample, and we use the 8 MSBs from the sample as the PWM value. The ADC takes 10-bit samples, but if you need only 8 bits, as we do here, you can drop the LSBs. This is equivalent to dividing by 4.

Finally, we connect the output of the PWM module to all eight LEDs.

Build and load the project. If you have a potentiometer, you can connect it between GND and *V+* and connect its output to *A0* to vary the brightness of the LEDs. Be careful not to connect it to RAW, as this will damage the microcontroller.

All the Channels

Now that we have one channel continuously sampling, we can make it a little more interesting by sampling all eight. To do this, we are going to use a counter that counts from 0–7. However, we have to modify the count value to generate the actual channel value. If the counter is greater than 1 (2–7), we will add 2 to it so it will count 0–1 then 4–9.

We need to increment the counter after we receive a sample for the currently selected channel. To do this, we need to convert the sample channel back to a counter value by subtracting 2 when the value is greater than 1.

To store these two intermediate values, we can use `sig` types:

```
sig real_channel[4];
sig sample_channel[3];
```

The `sig real_channel` will hold the ADC channel that the counter is on, and `sample_channel` will be the 0–7 value converted from the `avr.sample_channel` value.

Generating these two values is pretty easy using the ternary operator. The ternary operator, `?`, selects one of two values based on the logical value of a statement. It looks like *`STATEMENT ? WHEN_TRUE : WHEN_FALSE`*. When *`STATEMENT`* is true, the first, *`WHEN_TRUE`*, expression will be used: and when it is false, the second, *`WHEN_FALSE`*, expression will be used:

```
real_channel = channel_ctr.q > 1 ? channel_ctr.q + 2 : channel_ctr.q;
sample_channel =
  avr.sample_channel > 1 ? avr.sample_channel - 2 : avr.sample_channel;

if (avr.new_sample && sample_channel == channel_ctr.q)
  channel_ctr.d = channel_ctr.q + 1;
```

When the value is greater than 1, we select the modified value. Otherwise, we pass the signal along unchanged.

Each time a new sample shows up, if it matches the counter value, we increment the counter.

Because we are going to have a different value for each LED, we need eight PWM modules. We could instantiate eight different ones, but we can instead instantiate an array of modules:

```
pwm pwm[8]; // PWM modules for the LEDs
```

When you create an array of modules, each port has the module's dimensions prepended to it. That means the signal `pwm.update`, which used to be a single bit, is now an array of 8 bits. If a signal was already an array, such as `pwm.value`, it becomes a multi-dimensional array. In this case, `pwm.value` is now an 8 x 8 array. The first index selects the module instance.

We can use the `sample_channel` value to index into the array of modules to select which one receives the new sample. However, we need to assign a default value to all the modules because seven of the eight won't be assigned a value now. We just need to set `pwm.update` to 0 and `pwm.value` to *don't care*, as it doesn't matter when `pwm.update` is 0:

```
pwm.update = 8h00;       // default values
pwm.value = 8x{{8hxx}};

// update when we have a new sample
pwm.update[sample_channel] = avr.new_sample;
pwm.value[sample_channel] = avr.sample[9:2]; // use the 8 MSBs

led = pwm.pulse;         // connect PWM output to all the LEDs
```

Remember that `pwm.value` is now an 8 x 8 array, so when we assign a default value, it also needs to be an 8 x 8 array. The value `8hxx` is an array of size 8, so we need to wrap it in the array builder syntax, `{}`, to make it a 1 x 8 array. We can then use the array duplication syntax, `Nx{}`, to duplicate the outer dimension eight times, making it an 8 x 8 array.

With the default values taken care of, we can use the `sample_channel` value to index the specific PWM module. Finally, because `pwm.pulse` is an 8-bit array, we can simply assign it to `led`. The full `mojo_top` source can be found in the "ADC Multichannel Example" on page 207.

If you build and load the project, the eight LEDs will now show the eight ADC inputs. Try connecting individual inputs to different voltages between 0 V and 3.3 V.

CHAPTER 9
A Basic Processor

In this chapter, we will design a basic processor and write some code to run on it. In their most basic form, processors aren't that complicated. They are more or less a programmable FSM. However, the latest processors from Intel or AMD are insanely complex devices that utilize many clever improvements over our basic design to squeeze out more performance. Look at this example as a starting point that you can modify and improve in the future.

Why even bother with a processor? Processors have three big advantages. The first is that you can drastically change their behavior by only changing the contents of a ROM. The exact same hardware can do many tasks. If you make your ROM external, the FPGA design doesn't even need to change for it to perform different tasks (like a computer). The second is that processors can easily handle sequential tasks. With just hardware, it can be complicated to coordinate complicated tasks that need to come in sequence. You usually end up with a lot of FSMs, which basically turn out to be non-programmable processors. The final and possibly biggest benefit is that processors are extremely resource efficient. This is because a processor has a single adder, multiplier, comparison circuit, and so forth, and each one is used every time your code calls for that operation. If you need to do two multiplications in a hardware pipeline (with each step of your task performed by a different stage in parallel), you need to instantiate two multipliers, taking up twice the resources. Granted, a full pipeline will drastically outperform a simple processor.

Just like everything else we have covered so far, processors are just another tool for your toolbox. In an FPGA, they can be incredibly helpful to orchestrate complex tasks, but are not an end-all tool. Otherwise, you wouldn't be bothering to learn hardware design, as you could just buy a better processor and write code.

What Is a Processor?

You are probably familiar with at least the idea of a processor. However, we need to be clear about our definition of a processor and what it should do.

In the most abstract form, a *processor* is a circuit whose behavior is determined by the set of instructions provided to it. This way, the same circuit can perform a completely different task simply by switching out the instructions. In general, a processor also has some form of memory to work with, and all processors need a way to input and output data. Without any external I/O, a processor can't do anything meaningful.

Instruction Sets

A processor's *instruction set* is the language that it speaks. It is the set of instructions that the processor understands. An *instruction* is a set of bits that has some meaning to the processor. We will be making up our own simple instruction set. There are many instruction sets out there, with the most famous being x86 (and its 64-bit extension, x86_64). This is the instruction set that computers with processors from Intel or AMD understand.

There are two important widths in a processor. The first is the one you hear about most, the *data path*. This is the largest chunk of data the processor can deal with. When you hear of an 8-bit processor, that means the processor can deal with 8-bit values. If you want it to work with bigger numbers, you have to play tricks and manipulate only 8 bits at a time. Most computer (desktop and laptop) processors are 64-bit.

The other important size is the *instruction size*. This is the number of bits each instruction is encoded with. These two sizes are often the same, but they don't have to be. Some instruction sets, like x86, even have variable-length instructions, but that complicates things quite a bit. For our processor, we are going to have an 8-bit data path with 16-bit instructions.

Memory

Before we dive into what our processor will do, we need to figure out how data will get in and out—and when it is in, how do we work with it? Processors typically have a tiny bit of super fast memory built into them. This memory is known as the processor's *registers* and serves as the working memory. Our processor will be able to perform operations only directly on these registers. So if all the work happens on these registers, how do we process outside data?

We need a way to load data into a register and output the value of a register. The interface we will use will be basically the same as the RAM interface in Chapter 6. We will specify an address and data (for a write), or we will specify an address and receive

data (for a read). This will be covered a bit more later. For now, just know that we have internal memory called registers and that we will have a way to load and store values to/from the outside world.

Initial Design

Before we can start to write any code, we need to decide on a set of instructions our processor will understand. We will be using 16 bits to encode each instruction, but we need to decide what to do with these bits.

We are going to use the first 4 bits for something known as the *opcode*, or operation code. This means we can have 16 different instructions, which will be plenty for now. Some of our instructions will require three arguments, which means we can divide the remaining 12 bits into 4-bit chunks for each argument. These arguments will be the addresses of registers, so we can have 16 registers.

NOP

What's the most basic instruction you can think of? An instruction that does nothing! This instruction is usually encoded as all 0s and known as *no operation*, or *NOP*:

NOP	0

Because this instruction does nothing, we can fill the remaining 12 bits with 0s.

LOAD and STORE

As you may remember from earlier, a processor needs a way to get data in and out of itself. If it can't output data, it doesn't matter what the processor does. We will create two more instructions, `LOAD` and `STORE`, to get data in and out, respectively:

LOAD	DEST	ADDR	OFFSET
STORE	SRC	ADDR	OFFSET

Because the loads and stores need to provide both a value and an address, we need two arguments. The `DEST` argument is the register that will get the value from a `LOAD`. The `SRC` argument is the register whose value will be output from a `STORE`. The `ADDR` argument is the register whose value should be used as the address. Finally, `OFFSET` is a constant that will be added to the value of the `ADDR` register to get the address. This offset can be convenient to have, but you don't really need it. I added it because there were 4 extra bits we could do something with.

SET

It is common to have to write a constant to a register. The `SET` instruction will let us do that:

SET	DEST	CONST

Here `DEST` is the register that will be set, and `CONST` is the 8-bit value to set it to. Keep in mind that because we are making an 8-bit processor, the registers are 8 bits wide.

LT and EQ

We will likely want to be able to compare two values. To do any comparison, we need only two operators, less than (`LT`) and equal (`EQ`). If we then wanted to perform a greater-than, we could use less than and flip the arguments. Less than or equal to can be achieved by using both:

LT	DEST	OP1	OP2
EQ	DEST	OP1	OP2

In the `LT` case, the `DEST` register will be 1 if `OP1` is less than `OP2`, and 0 otherwise. `EQ` is the same, except it checks for equality.

BEQ and BNEQ

Typically, a program will execute instruction after instruction. However, it can be powerful to be able to control the flow. This is where branching instructions are used:

BEQ	OP1	CONST
BNEQ	OP1	CONST

If register `OP1` is equal to the value `CONST`, then the `BEQ` instruction will skip the instruction following it. Otherwise, the program will continue as normal. `BNEQ` does the same thing but skips if they are not equal.

These instructions allow you to create `if` statements in your code.

The ALU

Processors commonly need to manipulate the values in the registers. Operations such as addition, AND, OR, bit shifting, and so forth, are all typically contained in some-

thing called the *arithmetic logic unit* (ALU). These types of instructions will make up the remainder of our instruction set:

ADD	DEST	OP1	OP2
SUB	DEST	OP1	OP2
SHL	DEST	OP1	OP2
SHR	DEST	OP1	OP2
AND	DEST	OP1	OP2
OR	DEST	OP1	OP2
INV	DEST	OP1	0
XOR	DEST	OP1	OP2

`ADD` and `SUB` add or subtract `OP1` and `OP2` and store the result into `DEST`.

`SHL` and `SHR` shift `OP1` to the right or left by `OP2` bits and store the result into `DEST`.

`AND`, `OR`, and `XOR` perform their respective bitwise operations on `OP1` and `OP2` and store the result into `DEST`.

`INV` does a bitwise inversion of `OP1` and stores the result into `DEST`.

The Program Counter

We are going to put all the instructions for whatever program we write into a ROM. The ROM will need an address in order to know what instruction we need. This address will be specified by the *program counter*. For our processor, we will use register 0 as our program counter. This will allow us to directly manipulate the flow of the program by messing with its value.

The Processor

Check out the full processor module at "Basic Processor" on page 209.

You may notice that the entire module including the constant declarations is under 100 lines long! It's pretty amazing how much functionality you can get with such a simple design.

Something you haven't seen yet is the use of a `global` block to declare constants:

```
global Inst {
  c NOP   = 4d0;  // 0 filled
  c LOAD  = 4d1;  // dest, op1, offset  : R[dest] = M[R[op1] + offset]
```

```
    c STORE = 4d2;  // src, op1, offset   : M[R[op1] + offset] = R[src]
    c SET   = 4d3;  // dest, c           : R[dest] = c
    c LT    = 4d4;  // dest, op1, op2    : R[dest] = R[op1] < R[op2]
    c EQ    = 4d5;  // dest, op1, op2    : R[dest] = R[op1] == R[op2]
    c BEQ   = 4d6;  // op1, c            : R[0] = R[0] + (R[op1] == c ? 2 : 1)
    c BNEQ  = 4d7;  // op1, c            : R[0] = R[0] + (R[op1] != c ? 2 : 1)
    c ADD   = 4d8;  // dest, op1, op2    : R[dest] = R[op1] + R[op2]
    c SUB   = 4d9;  // dest, op1, op2    : R[dest] = R[op1] - R[op2]
    c SHL   = 4d10; // dest, op1, op2    : R[dest] = R[op1] << R[op2]
    c SHR   = 4d11; // dest, op1, op2    : R[dest] = R[op1] >> R[op2]
    c AND   = 4d12; // dest, op1, op2    : R[dest] = R[op1] & R[op2]
    c OR    = 4d13; // dest, op1, op2    : R[dest] = R[op1] | R[op2]
    c INV   = 4d14; // dest, op1         : R[dest] = ~R[op1]
    c XOR   = 4d15; // dest, op1, op2    : R[dest] = R[op1] ^ R[op2]
}
```

The constant declarations are like any we used before inside the module declaration. But, when they are declared in a `global` block, they become accessible throughout our entire design.

These are the opcodes for each of the instructions. The actual value for each code doesn't particularly matter (except it is convenient if `NOP` is 0). What does matter is that we use consistent encoding throughout our design. Accessing these is similar to accessing a globally declared struct, with *`Namespace.CONST`*—for example, `Inst.LOAD`.

The processor's registers are nothing more than a 2D array of DFFs:

```
    .clk(clk), .rst(rst) {
      dff reg[16][8]; // CPU Registers
    }
```

We have 16 registers of 8 bits each. Using an array of DFFs makes it easy to access multiple registers in the same cycle, but it wouldn't scale well if the bank was huge:

```
  instRom instRom;  // program ROM
```

We can't forget our program ROM! We will cover the actual ROM in a bit. However, it has only two ports, `address` and `inst`. It outputs the corresponding instruction on `inst` for the given `address`. It works the same as the ROM in Chapter 6.

The design also uses five signals for intermediate values:

```
    sig op[4];        // opcode
    sig arg1[4];      // first arg
    sig arg2[4];      // second arg
    sig dest[4];      // destination arg
    sig constant[8];  // constant arg
```

These will be used to rename parts of the instruction so that the rest of the design is easier to read. They aren't strictly needed, as we could just pull the bits out of the instruction whenever we need them. Here are the actual assignments:

```
op = instRom.inst[15:12];     // opcode is top 4 bits
dest = instRom.inst[11:8];    // dest is next 4 bits
arg1 = instRom.inst[7:4];     // arg1 is next 4 bits
arg2 = instRom.inst[3:0];     // arg2 is last 4 bits
constant = instRom.inst[7:0]; // constant is last 8 bits
```

The way we defined our instruction set, the *opcode* is always the 4 most significant bits. The other four signals are just the common names for the different parts of the instruction.

Remember we said that we would use register 0 as the program counter? Here are the two lines that use it:

```
instRom.address = reg.q[0];   // reg 0 is program counter
reg.d[0] = reg.q[0] + 1;      // increment PC by default
```

We use register 0 to index into the program ROM, and by default it will increment by 1 each cycle. Incrementing it keeps the program moving along. If the program writes a value to it, it will receive that value instead of the increment.

Finally, we have the meat of the processor:

```
// Perform the operation
case (op) {
  Inst.LOAD:
    read = 1;                                  // request a read
    reg.d[dest] = din;                         // save the data
    address = reg.q[arg1] + arg2;              // set the address
  Inst.STORE:
    write = 1;                                 // request a write
    dout = reg.q[dest];                        // output the data
    address = reg.q[arg1] + arg2;              // set the address
  Inst.SET:
    reg.d[dest] = constant;                    // set the reg to constant
  Inst.LT:
    reg.d[dest] = reg.q[arg1] < reg.q[arg2];   // less than comparison
  Inst.EQ:
    reg.d[dest] = reg.q[arg1] == reg.q[arg2];  // equals comparison
  Inst.BEQ:
    if (reg.q[dest] == constant)               // if R[dest] == constant
      reg.d[0] = reg.q[0] + 2;                 // skip next instruction
  Inst.BNEQ:
    if (reg.q[dest] != constant)               // if R[dest] != constant
      reg.d[0] = reg.q[0] + 2;                 // skip next instruction
  Inst.ADD:
    reg.d[dest] = reg.q[arg1] + reg.q[arg2];   // addition
  Inst.SUB:
    reg.d[dest] = reg.q[arg1] - reg.q[arg2];   // subtraction
  Inst.SHL:
    reg.d[dest] = reg.q[arg1] << reg.q[arg2];  // shift left
  Inst.SHR:
    reg.d[dest] = reg.q[arg1] >> reg.q[arg2];  // shift right
  Inst.AND:
```

```
        reg.d[dest] = reg.q[arg1] & reg.q[arg2];   // bitwise AND
      Inst.OR:
        reg.d[dest] = reg.q[arg1] | reg.q[arg2];   // bitwise OR
      Inst.INV:
        reg.d[dest] = ~reg.q[arg1];                // bitwise invert
      Inst.XOR:
        reg.d[dest] = reg.q[arg1] ^ reg.q[arg2];   // bitwise XOR
    }
```

It is really just one big `case` statement performing a operation for each different opcode.

The Program

Now that we have a processor, we need a program for it to run. Take a look at this example `instRom`:

```
module instRom (
    input address[8],
    output inst[16]
  ) {

  always {
    inst = c{Inst.NOP, 12b0};

    case (address) {
      // begin:
      0: inst = c{Inst.SET, 4d2, 8d0};              // SET R2, 0
      // loop:
      1: inst = c{Inst.SET, 4d1, 8d128};            // SET R1, 128
      2: inst = c{Inst.STORE, 4d2, 4d1, 4d0};       // STORE R2, R1, 0
      3: inst = c{Inst.SET, 4d1, 8d1};              // SET R1, 1
      4: inst = c{Inst.ADD, 4d2, 4d2, 4d1};         // ADD R2, R2, R1
      5: inst = c{Inst.SET, 4d15, 8d1};             // SET R15, loop
      6: inst = c{Inst.SET, 4d0, 8d7};              // SET R0, delay
      // delay:
      7: inst = c{Inst.SET, 4d11, 8d0};             // SET R11, 0
      8: inst = c{Inst.SET, 4d12, 8d0};             // SET R12, 0
      9: inst = c{Inst.SET, 4d13, 8d0};             // SET R13, 0
      10: inst = c{Inst.SET, 4d1, 8d1};             // SET R1, 1
      // delay_loop:
      11: inst = c{Inst.ADD, 4d11, 4d11, 4d1};      // ADD R11, R11, R1
      12: inst = c{Inst.BEQ, 4d11, 8d0};            // BEQ R11, 0
      13: inst = c{Inst.SET, 4d0, 8d11};            // SET R0, delay_loop
      14: inst = c{Inst.ADD, 4d12, 4d12, 4d1};      // ADD R12, R12, R1
      15: inst = c{Inst.BEQ, 4d12, 8d0};            // BEQ R12, 0
      16: inst = c{Inst.SET, 4d0, 8d11};            // SET R0, delay_loop
      17: inst = c{Inst.ADD, 4d13, 4d13, 4d1};      // ADD R13, R13, R1
      18: inst = c{Inst.BEQ, 4d13, 8d0};            // BEQ R13, 0
      19: inst = c{Inst.SET, 4d0, 8d11};            // SET R0, delay_loop
      20: inst = c{Inst.SET, 4d1, 8d0};             // SET R1, 0
      21: inst = c{Inst.ADD, 4d0, 4d15, 4d1};       // ADD R0, R15, R1
```

```
    }
  }
}
```

This code will slowly increment a counter and output its value to address 128. Before we dive into the details of the ROM, let's hook everything up and display the output value on the LEDs:

```
module mojo_top (
    input clk,              // 50MHz clock
    input rst_n,            // reset button (active low)
    output led [8],         // 8 user controllable LEDs
    input cclk,             // configuration clock, AVR ready when high
    output spi_miso,        // AVR SPI MISO
    input spi_ss,           // AVR SPI Slave Select
    input spi_mosi,         // AVR SPI MOSI
    input spi_sck,          // AVR SPI Clock
    output spi_channel [4], // AVR general purpose pins
    input avr_tx,           // AVR TX (FPGA RX)
    output avr_rx,          // AVR RX (FPGA TX)
    input avr_rx_busy       // AVR RX buffer full
  ) {

  sig rst;                  // reset signal

  .clk(clk) {
    // The reset conditioner is used to synchronize the reset signal to the FPGA
    // clock. This ensures the entire FPGA comes out of reset at the same time.
    reset_conditioner reset_cond;

    .rst(rst) {
      cpu cpu;        // our snazzy CPU
      dff led_reg[8]; // storage for LED value
    }
  }

  always {
    reset_cond.in = ~rst_n;   // input raw inverted reset signal
    rst = reset_cond.out;     // conditioned reset

    spi_miso = bz;            // not using SPI
    spi_channel = bzzzz;      // not using flags
    avr_rx = bz;              // not using serial port

    cpu.din = 8hxx;           // default to don't care

    // if cpu uses address 128
    if (cpu.address == 128) {
      if (cpu.write)
        led_reg.d = cpu.dout; // update the LED value
      if (cpu.read)
        cpu.din = led_reg.q;  // let the CPU read the LED value
```

```
    }

    led = led_reg.q;          // connect LEDs to led_reg
  }
}
```

Notice that this design is similar to the Register Interface example, where we received write and read requests for a given address. The big difference here is that we have to provide the read value immediately. Before, we had up to a quarter second to generate the requested value. We could change the processor's design so that it will stall as it waits for a read, but that is an enhancement you can add.

So why did we use address 128 for the LEDs instead of address 0? Conventionally, the first half of the memory space is reserved for RAM, and the second half is reserved for memory-mapped I/O. *Memory-mapped I/O* is a term used to describe the address-based interface we are using. The addresses of reads and writes don't go to RAM but rather to peripherals allowing the processor to interact with the outside world. This is how your computer works. The drivers you install for new hardware are used as the glue from a set of special registers to some simpler abstraction.

You should now be able to build and load the project to see the LEDs counting. This example is simple (counting LEDs could have been done long ago), but you can now change what the LEDs do simply by changing the contents of the ROM.

The Assembler

If you try to edit that ROM directly, it would be a bit of nightmare because of all the hardcoded addresses. If you need to insert an instruction, you have to renumber every instruction after it and change any instruction number constants. Instead, we can use something known as an *assembler* to make life easier.

An assembler takes assembly code and turns it into machine code (our ROM). Assembly is just barely an abstraction above machine code, as it has a one-to-one relationship, but it makes it much easier to read and edit.

The assembly I used to generate that ROM is shown here:

```
begin:
  SET R2, 0           // R2 = 0
loop:
  SET R1, 128         // R1 = 128
  STORE R2, R1, 0     // M[128] = R2
  SET R1, 1           // R1 = 1
  ADD R2, R2, R1      // R2++
  SET R15, loop       // R15 = loop, R15 is return address
  SET R0, delay       // goto delay
delay:
  SET R11, 0          // R11 = 0
  SET R12, 0          // R12 = 0
```

```
    SET R13, 0          // R13 = 0
    SET R1, 1           // R1 = 1
  delay_loop:
    ADD R11, R11, R1    // R11++
    BEQ R11, 0          // skip next if (R11 == 0)
    SET R0, delay_loop  // goto delay_loop
    ADD R12, R12, R1    // R12++
    BEQ R12, 0          // skip next if (R12 == 0)
    SET R0, delay_loop  // goto delay_loop
    ADD R13, R13, R1    // R13++
    BEQ R13, 0          // skip next if (R13 == 0)
    SET R0, delay_loop  // goto delay_loop
    SET R1, 0           // R1 = 0
    ADD R0, R15, R1     // R0 = R15 (return)
```

This should be a little easier to read and understand than the raw ROM from before.

Although our CPU treats only R0 specially (it's the program counter), it is helpful to assign special roles to some registers for use in our programs. For example, for many instructions, it can be handy to have a temporary register to load a constant into. I used R1 for this purpose. The other special register I used was R15 to store the return address from a function call (more on this later).

The goal of this program is to count and output the count to address 128. Because we have no external RAM, all our memory needs to fit in the 16 registers. I chose R2 to store our primary counter.

The first line initializes R2 to 0; we then enter the main loop. The main loop starts by writing R2 to address 128. R1 is used to store the address 128 used by the `STORE` instruction. We then increment R2 by 1. Again, we use R1 to store the constant 1 to add to R2.

We could simply loop here, but it would count way too fast for us to see on the LEDs. So we need to create a delay function. A *function* is just a block of code that can be called from anywhere and will return to where it's called from. We use R15 to specify the address we want to return to. In our case, we cheat a little bit and set R15 to the beginning of our loop instead of the instruction after the call to `delay`. This is just a little more efficient.

We can use the `SET` instruction with R0 to jump to anywhere in our program. This is where labels are really helpful, since we don't care what the actual instruction number is because the label will get replaced with the proper value.

The delay function uses R11, R12, and R13 to count from 0 to 16,777,215. This takes a decent amount of time, so we can actually see the LEDs change. When the counter overflows, the delay function returns to the address in R15 by using the `ADD` instruction to set R0 to R15.

So how do we convert the assembly code to our ROM? I've written a Java program that does this and is available on GitHub or from Embedded Micro.

If you don't want to modify the assembler, you can download the *skinny-asm.jar* file in the root of the repository. You can then use the JAR from the command line as follows:

```
java -jar skinny-asm.jar assembly-file.asm
```

This should turn *assembly-file.asm* into the `instRom` module, or tell you what is wrong with your assembly.

Try editing this code, or write you own program! Feel free to even swap out some of the instructions for your own if you can think of something more useful. Maybe you'd prefer an instruction for directly setting a bit instead of XOR, or maybe you would rather be able to tell whether a value is even or odd instead of shifting right.

Congratulations on making your first processor!

CHAPTER 10
FPGA Internals

In this chapter, we are going to briefly cover the internals of an FPGA and how they work. This chapter provides an overview of how the different parts work with enough detail that you can then use this knowledge to help better inform your design decisions. However, we will stay out of the nitty-gritty.

General Fabric and Routing Resources

The biggest part of the FPGA is something known as the *general fabric*. This is where most of your design gets implemented. It consists of a bunch of pieces known as *configurable logic blocks*, or CLBs. In the case of the FPGA used on the Mojo, the Spartan 6, a CLB contains a few six-input LUTs, eight flip-flops, and some have extra resources such as RAM or special routing for carry logic (like the arbiter).

The way all the combinational logic in your designs gets implemented is in lookup tables. The Spartan 6 uses six input LUTs. Each one of these can have a completely unique mapping of any six inputs to a single output. This means each LUT can implement any six-to-one function regardless of its complexity. For added flexibility, the LUTs in this FPGA can also be used as two five-input LUTs (using the same five inputs).

All these LUTs and DFFs are connected to something known as the *switch matrix*. This is a basically what the name indicates: a huge matrix of switches. It is how specific CLBs and other resources are connected to each other. There aren't too many details available about what exactly the switch matrix consists of or how it is constructed (a lot of the FPGA's internals are proprietary), but you can imagine it as a bunch of wires and multiplexers to control how the wires are connected.

There are some special routing resources in the switch matrix for special signals such as clocks, but most of the time the tools will figure this out for you and route things

accordingly. This is why you can run into trouble when you try to use a clock signal for something other than a clock, or a signal that isn't a clock as a clock.

FPGAs are basically a bunch of LUTs and DFFs that can be connected however they need to be. Let's look a little closer at the details of the Spartan 6.

CLBs contain two slices. There are three types of slices: *SLICEX*, *SLICEL*, and *SLICEM*. SLICEX is the simplest of them, and SLICEM is the most complex. Every slice has four LUTs and eight flip-flops.

The SLICEL slices contain everything in a SLICEX but with routing logic for carry chains. The SLICEM slices have everything in a SLICEL but also can use their LUTs as small RAMs or shift registers.

The FPGA on the Mojo has 1,430 slices: 360 SLICEMs, 355 SLICELs, and 715 SLICEXs. This provides 5,720 LUTs, up to 90 Kb of distributed RAM, up to 45 Kb of shift registers, and 11,440 flip-flops for your designs.

For schematics of the slice types, see pages 9–11 of the Xilinx doc UG384.

What does this look like inside the FPGA? Xilinx's tool, PlanAhead, will show you the floor plan of your implemented design. Figure 10-1 shows two designs. The one on the left is from Chapter 9, while the one on the right is from a later project in Chapter 12. The lighter blue represents resources that are being used, and as you can tell, the simple processor has a fairly low utilization (~18%), while the other example uses most of the FPGA's slices (~96%).

In this image, the darker blue section that makes up most of the FPGA is the general fabric. This is where all the slices live. The dark red rectangles sandwiched in the general fabric are block RAMs. The green rectangles, also mixed in the fabric, are multipliers (DSP48A1). The orange rectangles in the middle are special clocking resources, and the colorful stuff around the edges represents I/O ports and I/O-related primitives. Some of these special primitives are covered in the next section.

You can view the floor plan for any of your built projects by launching PlanAhead and opening the PlanAhead project file generated by the Mojo IDE. These can be found in *WORKSPACE/PROJECT/work/planAhead/PROJECT/PROJECT.ppr*, where *WORKSPACE* is your workspace directory and *PROJECT* is the name of your project. With the project open, click Open Implemented Design on the left side. You should note that the whole PlanAhead project is deleted each time you build your project in the Mojo IDE, so any changes you make in PlanAhead won't persist.

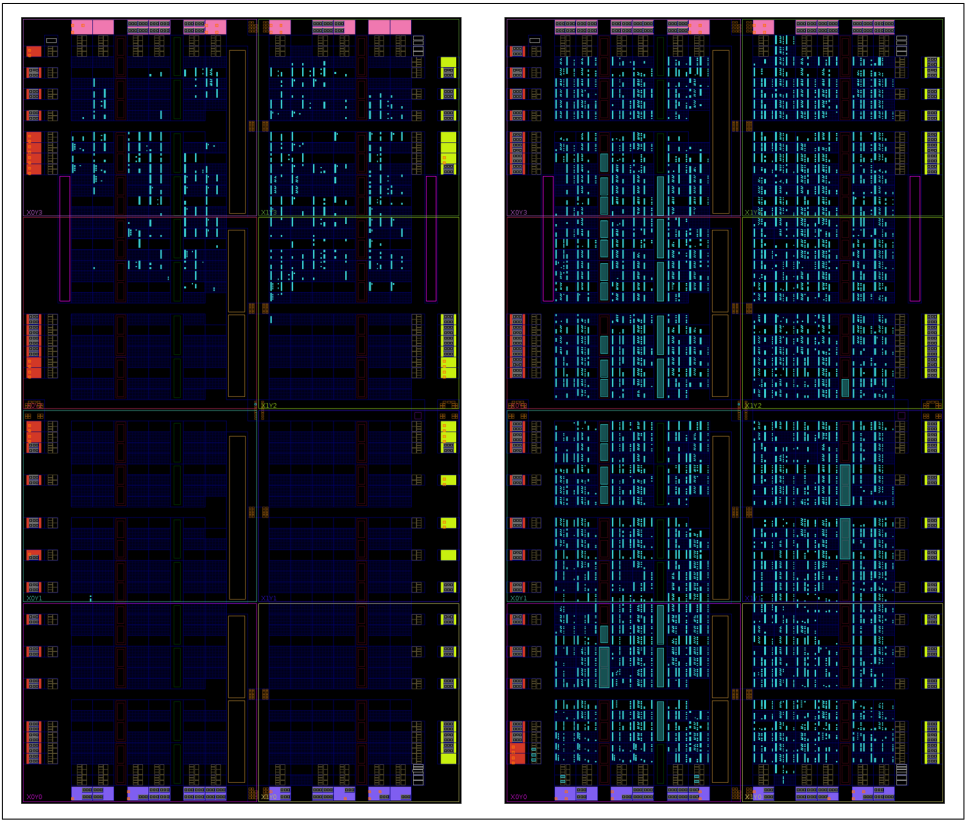

Figure 10-1. FPGA floor plans of the simple processor (left) and a more complicated project (right)

Special Primitives

Besides the general fabric, FPGAs typically contain other special *primitives* to efficiently perform certain functions. A lot of these primitives are automatically used to implement your design efficiently, while some you need to explicitly use.

In the following sections, we will touch on the primitives related to memory, math, clocking, and I/O. There are tons of primitives, but we will be looking only at the most commonly used ones. For a more complete list, see the "Spartan 6 Libraries Guide for HDL Designs."

Block RAM (Memory)

When you build your project, if you scroll up a little in the output after it is done building, you will find something like this:

```
Device Utilization Summary:

Slice Logic Utilization:
  Number of Slice Registers:                     97 out of  11,440    1%
    Number used as Flip Flops:                   97
    Number used as Latches:                       0
    Number used as Latch-thrus:                   0
    Number used as AND/OR logics:                 0
  Number of Slice LUTs:                        167 out of   5,720    2%
    Number used as logic:                      157 out of   5,720    2%
      Number using O6 output only:             123
      Number using O5 output only:              12
      Number using O5 and O6:                   22
      Number used as ROM:                        0
    Number used as Memory:                       5 out of   1,440    1%
      Number used as Dual Port RAM:              0
      Number used as Single Port RAM:            4
        Number using O6 output only:             0
        Number using O5 output only:             0
        Number using O5 and O6:                  4
      Number used as Shift Register:             1
        Number using O6 output only:             1
        Number using O5 output only:             0
        Number using O5 and O6:                  0
    Number used exclusively as route-thrus:      5
      Number with same-slice register load:      4
      Number with same-slice carry load:         1
      Number with other load:                    0

Slice Logic Distribution:
  Number of occupied Slices:                     59 out of   1,430    4%
  Number of MUXCYs used:                         16 out of   2,860    1%
  Number of LUT Flip Flop pairs used:           171
    Number with an unused Flip Flop:             85 out of     171   49%
    Number with an unused LUT:                    4 out of     171    2%
    Number of fully used LUT-FF pairs:           82 out of     171   47%
    Number of slice register sites lost
      to control set restrictions:                0 out of  11,440    0%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.

IO Utilization:
  Number of bonded IOBs:                         22 out of     102   21%
    Number of LOCed IOBs:                        22 out of      22  100%
    IOB Flip Flops:                               4

Specific Feature Utilization:
  Number of RAMB16BWERs:                         0 out of      32    0%
```

```
Number of RAMB8BWERs:                          0 out of      64    0%
Number of BUFIO2/BUFIO2_2CLKs:                 0 out of      32    0%
Number of BUFIO2FB/BUFIO2FB_2CLKs:             0 out of      32    0%
Number of BUFG/BUFGMUXs:                       1 out of      16    6%
  Number used as BUFGs:                        1
  Number used as BUFGMUX:                      0
Number of DCM/DCM_CLKGENs:                     0 out of       4    0%
Number of ILOGIC2/ISERDES2s:                   3 out of     200    1%
  Number used as ILOGIC2s:                     3
  Number used as ISERDES2s:                    0
Number of IODELAY2/IODRP2/IODRP2_MCBs:         0 out of     200    0%
Number of OLOGIC2/OSERDES2s:                   1 out of     200    1%
  Number used as OLOGIC2s:                     1
  Number used as OSERDES2s:                    0
Number of BSCANs:                              0 out of       4    0%
Number of BUFHs:                               0 out of     128    0%
Number of BUFPLLs:                             0 out of       8    0%
Number of BUFPLL_MCBs:                         0 out of       4    0%
Number of DSP48A1s:                            0 out of      16    0%
Number of ICAPs:                               0 out of       1    0%
Number of MCBs:                                0 out of       2    0%
Number of PCILOGICSEs:                         0 out of       2    0%
Number of PLL_ADVs:                            0 out of       2    0%
Number of PMVs:                                0 out of       1    0%
Number of STARTUPs:                            0 out of       1    0%
Number of SUSPEND_SYNCs:                       0 out of       1    0%
```

This is from the example project that asks for your name and says hello.

The first section of the report gives you an overall feel for how much of the FPGA you are using. The number of slice registers is how many DFFs you are using, only 97/11,440 in this case. The number of slice LUTs gives you a feel for how complex the logic in your design is, only 167/5,720 here. This decently complex project is using less than 2% of the FPGA's general fabric!

As you may remember, we used a special RAM component, with the reasoning being that the tools would realize this component was RAM and would utilize the RAM built into the FPGA. However, if you look under the Specific Feature Utilization category, both `RAMB16BWERs` and `RAMB8BWERs` are not being used at all (these are the block RAM primitives)!

This is because the RAM we used was set to be only 8 bits wide and 32 entries deep. This isn't very big, so the tools decided to put the RAM in the general fabric. If you look under the Number of Slices LUTs section, the Number Used as Single Port RAM shows that four LUTs were used to make the small RAM. We are actually fully utilizing the four LUTs here. This is because the four LUTs are operating as RAM with two outputs; all four are using both *O5* and *O6* outputs. The LUTs generally hold 64 entries (remember, they are 6-bit LUTs), but can be split to two LUTs of 5 bits each.

If we change the size of the RAM in our design to be 64 entries deep, we see the following change:

```
    Number used as Memory:                      9 out of   1,440    1%
      Number used as Dual Port RAM:             0
      Number used as Single Port RAM:           8
        Number using O6 output only:            8
        Number using O5 output only:            0
        Number using O5 and O6:                 0
```

Now we are using eight LUTs as RAM, but each one is using only one output, so each is 64 entries deep.

What happens if we crank up the size of the RAM to be 128 entries? The simple LUT RAM can go up to only 64 entries. Take a look at the following report:

```
Device Utilization Summary:

Slice Logic Utilization:
  Number of Slice Registers:                   89 out of  11,440    1%
    Number used as Flip Flops:                 89
    Number used as Latches:                     0
    Number used as Latch-thrus:                 0
    Number used as AND/OR logics:               0
  Number of Slice LUTs:                       140 out of   5,720    2%
    Number used as logic:                     138 out of   5,720    2%
      Number using O6 output only:            105
      Number using O5 output only:             12
      Number using O5 and O6:                  21
      Number used as ROM:                       0
    Number used as Memory:                      1 out of   1,440    1%
      Number used as Dual Port RAM:             0
      Number used as Single Port RAM:           0
      Number used as Shift Register:            1
        Number using O6 output only:            1
        Number using O5 output only:            0
        Number using O5 and O6:                 0
    Number used exclusively as route-thrus:     1
      Number with same-slice register load:     0
      Number with same-slice carry load:        1
      Number with other load:                   0

Slice Logic Distribution:
  Number of occupied Slices:                   51 out of   1,430    3%
  Number of MUXCYs used:                       16 out of   2,860    1%
  Number of LUT Flip Flop pairs used:         154
    Number with an unused Flip Flop:           72 out of     154   46%
    Number with an unused LUT:                 14 out of     154    9%
    Number of fully used LUT-FF pairs:         68 out of     154   44%
    Number of slice register sites lost
      to control set restrictions:              0 out of  11,440    0%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
```

```
    one Flip Flop within a slice.  A control set is a unique combination of
    clock, reset, set, and enable signals for a registered element.
    The Slice Logic Distribution report is not meaningful if the design is
    over-mapped for a non-slice resource or if Placement fails.

  IO Utilization:
    Number of bonded IOBs:                        22 out of     102   21%
      Number of LOCed IOBs:                       22 out of      22  100%
      IOB Flip Flops:                              4

  Specific Feature Utilization:
    Number of RAMB16BWERs:                        0 out of      32    0%
    Number of RAMB8BWERs:                         1 out of      64    1%
    Number of BUFIO2/BUFIO2_2CLKs:                0 out of      32    0%
    Number of BUFIO2FB/BUFIO2FB_2CLKs:            0 out of      32    0%
    Number of BUFG/BUFGMUXs:                      1 out of      16    6%
      Number used as BUFGs:                       1
      Number used as BUFGMUX:                     0
    Number of DCM/DCM_CLKGENs:                    0 out of       4    0%
    Number of ILOGIC2/ISERDES2s:                  3 out of     200    1%
      Number used as ILOGIC2s:                    3
      Number used as ISERDES2s:                   0
    Number of IODELAY2/IODRP2/IODRP2_MCBs:        0 out of     200    0%
    Number of OLOGIC2/OSERDES2s:                  1 out of     200    1%
      Number used as OLOGIC2s:                    1
      Number used as OSERDES2s:                   0
    Number of BSCANs:                             0 out of       4    0%
    Number of BUFHs:                              0 out of     128    0%
    Number of BUFPLLs:                            0 out of       8    0%
    Number of BUFPLL_MCBs:                        0 out of       4    0%
    Number of DSP48A1s:                           0 out of      16    0%
    Number of ICAPs:                              0 out of       1    0%
    Number of MCBs:                               0 out of       2    0%
    Number of PCILOGICSEs:                        0 out of       2    0%
    Number of PLL_ADVs:                           0 out of       2    0%
    Number of PMVs:                               0 out of       1    0%
    Number of STARTUPs:                           0 out of       1    0%
    Number of SUSPEND_SYNCs:                      0 out of       1    0%
```

We are now using no LUTs as RAM! Instead, notice that we are using `RAMB8BWER`. Because we used the special RAM component in our design, the tools knew that we wanted RAM and, depending on the size, they will use block RAM (as they are now) or distributed RAM (as before).

Something else to notice is that the numbers of registers and LUTs used for this design are lower than when the RAM was smaller by a fairly significant margin. This is because a lot of the logic needed before is built into the `RAMB8BWER` primitive. The tools will generally avoid using block RAM if it can, though, because we have thousands of registers and LUTs but only 64 `RAMB8BWERs` and 32 `RAMB16BWERs`.

Math

One of the more important primitives is `DSP48A1`. The FPGA on the Mojo has only 16 of these, but they are very important if you start doing any serious math in your design.

As we've seen, some of the slices are specifically designed to efficienctly implement circuits with a carry chain, such as addition and subtraction. This makes these operations cheap to implement in the general fabric. However, multiplication is a different beast. Each `DSP48A1` contains an 18-bit pre-adder/subtracter followed by an 18 x 18 signed multiplier and, finally, a 48-bit post-adder/subtracter. All of these operations are performed directly instead of having to go through the general fabric of the FPGA, making them substantially faster.

If you use multiplication in your design, keep an eye on this utilization number. If you exceed the number of `DSP48A1s` on the FPGA, the tools will have to put the multiplication logic into the general fabric. Depending on your clock speed and multiplication size, your design may start to fail timing.

Clocking

The primitives `DCM/DCM_CLKGEN` and `PLL_ADV` are used to synthesize new clock frequencies. This means generating new clocks from the single 50 MHz clock present on the Mojo. In some projects, you may find that 50 MHz isn't enough or that you may need another frequency for some high-speed I/O. You may even want a slower clock to save power. This can all be done with the clocking primitives.

These primitives are pretty complicated, and although you can manually instantiate them, it is a lot easier to use Xilinx's CoreGen tool. This tool is integrated into the Mojo IDE to make it easy to add cores to your project.

CoreGen is similar to the Components Library, except many of the cores are given to you as block boxes instead of open source. Some of the more complicated cores require you to buy a license to use them. However, most of the most commonly used basic ones are free.

To use CoreGen in your project, go to Project → Launch CoreGen. After a short delay, the CoreGen window should open, as shown in Figure 10-2. There is a tree in the left panel with all the cores you can generate. Expand the FPGA Features and Design category. Then expand the Clocking subcategory. Figure 10-2 shows the expanded category.

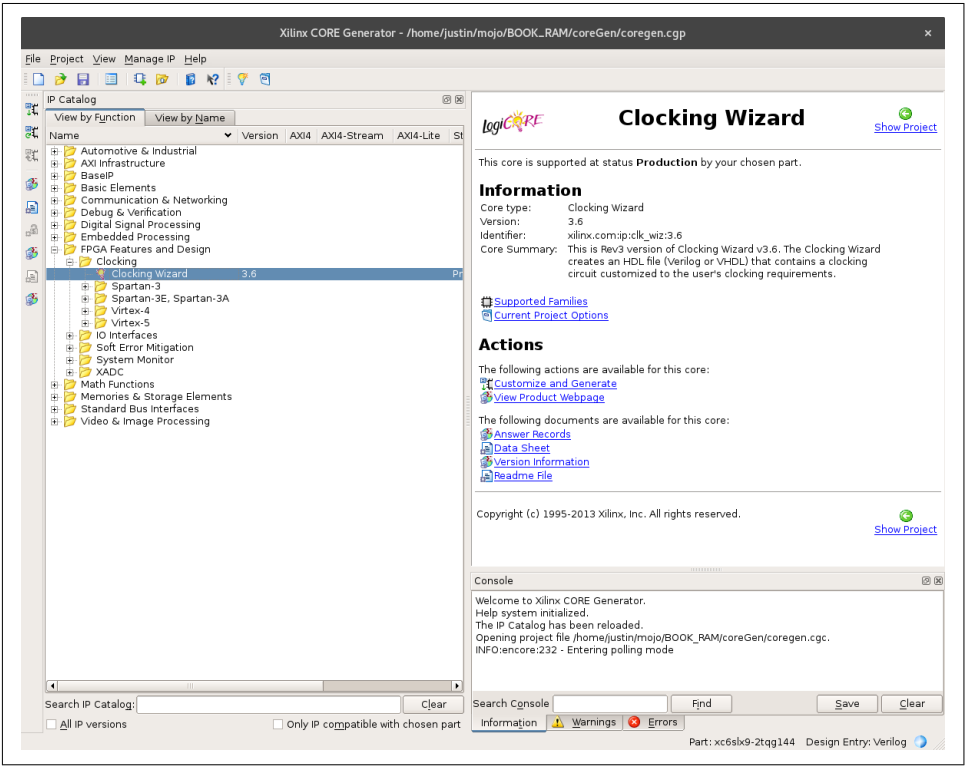

Figure 10-2. Xilinx's CoreGen tool

Double-click Clocking Wizard. A new window opens that will allow you to customize the core. On the first page, you can change the name of the module GoreGen will generate. By default, it is `clk_wiz_v3_6`, which is a bit uninspiring. On the same page, you also set the input clock frequency. By default, this is set to 100 MHz, but the Mojo has a 50 MHz clock, so make sure to change this. These settings are shown in Figure 10-3

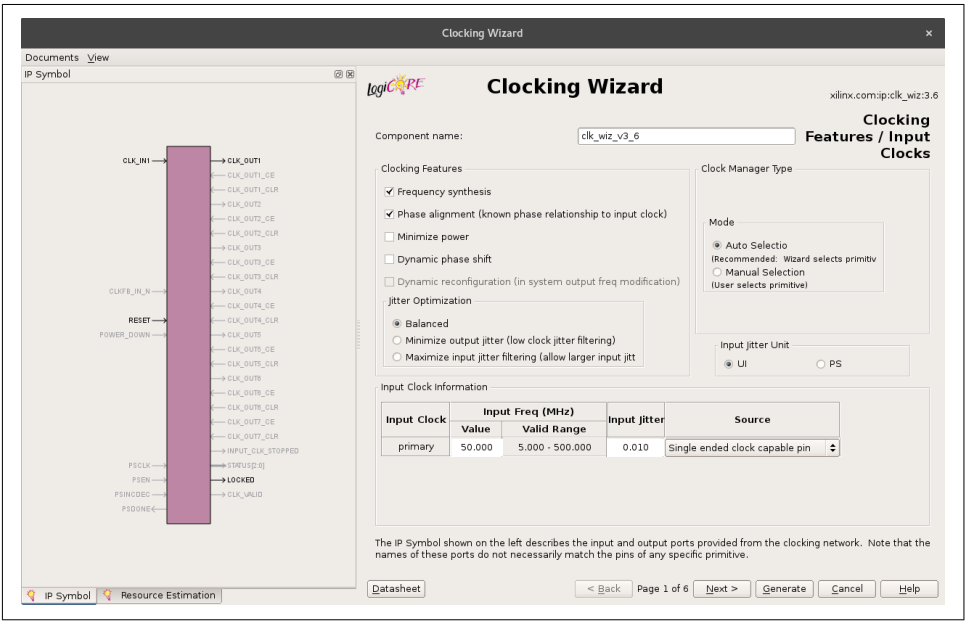

Figure 10-3. Xilinx's Clocking Wizard

Click the Next button to get to the next page. Here you can specify the new clock frequencies you want from the core. You can request up to six frequencies to be generated. However, note that mixing frequencies in your design requires a lot of care, as you can easily violate timing, and the tools won't hold your hand. When you request a frequency, the frequency column will tell you what the core will synthesize. This is because the FPGA can only multiply and divide the frequency by integers. So if you want 75 MHz, the FPGA could take the 50 Mhz, multiply it by 3, and then divide it by 2 to get exactly 75 Mhz. However, if you want 74 MHz, the best it can do is 73.077 MHz (50 x 19/13).

On the next page, you can select whether you want to use `RESET` and `LOCKED` signals. The reset signal stops all the output clocks and causes the circuit to have to reinitialize. The locked signal tells you when the output clocks are stable and at the requested frequency. Initially, it may take a small amount of time for this to happen. I generally don't use either of these signals, but if your design is highly dependent on clock frequency at startup, you may want to use the locked signal to hold the rest of your design in reset.

You can skip page 4 of the wizard. It just shows the parameters that were autogenerated.

Page 5 lets you rename the input and output ports. I usually like to rename the output clocks to indicate what they are being used for or include their frequency—for example, `clk_75` for a 75 MHz clock.

The last page is a summary of your settings. At this point, you can click Generate to make the core. After the core is done generating, a little readme window will pop up that you can close. You should now see your core in the CoreGen window in the bottom-left corner. You can close CoreGen at this point.

The Mojo IDE automatically detects when you generate a new core and will add it to your project under the Cores branch of your project tree. In the case of the Clocking Wizard core, the IDE simply generates some Verilog code to instantiate the primitives it needs. You can check out the source in the *Cores* directory. It's really not that readable, but it can be helpful to look at to get the names of the module's ports. You can now use this module like any other module in your design.

When generating the core or looking at its source, you may have noticed the `BUFG` primitive. This stands for *buffer global*, and is used to move a signal from the general fabric to a special global routing layer. This global network is generally reserved for clock signals, as it is designed to get a signal throughout the chip with low latency. Signals on this layer can't be used in the same way as other signals. For example, you can't connect a signal from the global network directly to an output pin.

In all our other designs, the tools automatically added `BUFGs` to the `clk` signal for us. You can see this in the utilization reports from before.

It can be easy to want to get carried away with the FPGA's clock speed. However, once you start cranking it up, meeting timing will become harder and harder. You shouldn't have too much trouble with a simple design running at 100–150 MHz, but if your design starts getting more complicated, making it workable will require a lot of effort. Just because you can generate a 375 MHz clock doesn't mean you should clock your design at the frequency.

Special I/O Features

The FPGA has a lot of primitives for performing special I/O functions. Two of the more impressive ones are `ISERDES` and `OSERDES`. These serialize or deserialize data from an input, or data to an output, respectively. For example, if you have an input running at 500 MHz, `ISERDES` can break this into four steams at 125 MHz for easier processing in the FPGA. `OSERDES` does the inverse, taking a few streams and combining them into a single high-speed signal. Each `SERDES` can do up to 1:4 or 4:1 ratios, but it is possible to combine two to get up to 1:8/8:1 ratios. The FPGA on the Mojo can use these to sample inputs or generate outputs up to 950 MHz.

Setting these up properly can be a bit difficult to figure out, especially when connecting two for higher ratios. Luckily, a core in CoreGen can help with this and other I/O tasks. You can find it under FPGA Features and Design → IO Interface → SelectIO Interface Wizard.

Another I/O feature is double-data rate inputs and outputs (`IDDR` and `ODDR`), where you input/output data on both the rising and falling edges of a clock (this works like a 1:2/2:1 [de]serializer). There is also the `IODELAY2` primitive that allows you to delay an input or output by tiny amounts of time to synchronize bus signals if the trace lengths on the PCB aren't the same length. Both of these are used directly via the SDRAM Controller in the Components Library.

Many of the FPGA-specific primitives can be used directly in Lucid by prefixing the names with `xil_`. For example, the `IODELAY2` primitive has the name `xil_IODELAY2` and can be used like any other module. See the SDRAM Component in the Mojo IDE's Components Library for an example of how these are used.

CHAPTER 11
Advanced Timing and Clock Domains

In this chapter, we will look at what happens when you break timing and some ways to fix the problem. We will also talk about having multiple clocks in your design and how to reliably get data between clock domains. This can be tricky because usually you can't ensure the that clocks are aligned or they aren't perfect multiples of each other. If you try to feed a signal synchronized to one clock into a DFF running off another clock, you will violate timing constraints periodically.

After finishing this chapter, you should know how to identify timing problems and know some techniques to fix them. This includes timing issues that the tools try to avoid when laying out and routing your design and issues that arise due to your design.

Breaking Timing and Fixing It with Pipelining

In this section, we are going to create a design that intentionally breaks timing by giving the tools an impossible task of performing a bunch of multiplications in a single clock cycle. We will identify the issue by looking at the timing report generated by the tools and then fix the design by splitting the operation into multiple clock cycles.

If you are using the 50 MHz clock on the Mojo without changing the frequency, you won't likely run into timing issues for your designs. However, it is still quite possible to break timing, and we'll take a look at a technique known as *pipelining* that can be used to help timing.

To demonstrate this, we need a design that fails timing. Create a new project based on the IO Shield Base example. Then create a new module named *timing.luc* with the following contents:

```
module timing (
    input clk,    // clock
```

```
    input a [8],  // first input
    input b [8],  // second input
    output c [64] // output
  ) {

  .clk(clk) {
    dff a_reg [8];  // register to hold a
    dff b_reg [8];  // register to hold b
    dff c_reg [64]; // register to hold c
  }

  always {
    a_reg.d = a; // connect inputs
    b_reg.d = b;
    c = c_reg.q; // connect output

    // super long expression of lots of multiplication
    c_reg.d = (a_reg.q * a_reg.q) * (a_reg.q * a_reg.q) *
      (b_reg.q * b_reg.q) * (b_reg.q * b_reg.q);
  }
}
```

To break timing, we can perform a bunch of multiplication. Because this is the only multiplication in our design, all the operations have `DSP48A1s` to use making it faster. If this was performed in the general fabric, a much simpler expression would break timing.

We can add this module to the top-level module and build the project:

```
  .clk(clk) {
    // The reset conditioner is used to synchronize the reset signal to the FPGA
    // clock. This ensures the entire FPGA comes out of reset at the same time.
    reset_conditioner reset_cond;

    timing timing;
  }

  always {
    reset_cond.in = ~rst_n; // input raw inverted reset signal
    rst = reset_cond.out;   // conditioned reset

    led = 8h00;             // turn LEDs off
    spi_miso = bz;          // not using SPI
    spi_channel = bzzzz;    // not using flags
    avr_rx = bz;            // not using serial port

    io_seg = 8hff;          // turn segments off
    io_sel = 4hf;           // select no digits

    timing.a = io_dip[0];
    timing.b = io_dip[1];
    io_led = {timing.c[16+:8], timing.c[8+:8], timing.c[0+:8]};
```

```
  }
}
```

Here we hook up the timing module's inputs and outputs to the switches and LEDs on the IO Shield. We need to use some nonconstant inputs and actually use the output so the module doesn't get optimized away. With external inputs and outputs, the tools can't assume anything.

Build the project. It will still generate a configuration file, but this file may not work. If you scroll up in the output, you'll see something like this:

```
------------------------------------------------------------------------------
  Constraint       | Check | Worst Case |  Best Case | Timing |   Timing
                   |       |    Slack   | Achievable | Errors |    Score
------------------------------------------------------------------------------
* TS_clk = PERIOD  | SETUP |    -5.748ns|    25.748ns|      82|      320922
  TIMEGRP "clk"
  50 MHz HIGH
   50%             | HOLD  |     3.995ns|            |       0|           0
------------------------------------------------------------------------------


1 constraint not met.
```

This tells you that the best it could do for this design was 25.748 ns, or 38.84 MHz. With a 50 MHz clock, 20 ns period, things won't work out. The timing score is the total time in picoseconds of all the signals that fail timing. 320922 is a pretty bad score. If this number is small and the worst-case slack is also small, the design likely will still work under most conditions, but it isn't guaranteed.

So how do we know where our design broke? In this case, it's trivial because we designed it to break in one spot, but if it is your own design, you might not know where it is failing. At the time of writing, the Mojo IDE doesn't have any tools to help with closing timing. Instead, we need to take a look at one of the log files generated when you built the project.

If you look in the project's folder, there should be a file with a similar path to *BreakTiming/work/planAhead/BreakTiming/BreakTiming.runs/impl_1/mojo_top_0.twr* (I named the project "BreakTiming"). This is the timing report. You can open this file in the Mojo IDE since it is just a text file. Just make sure to select the * filter for file types in the open file dialog.

There are 80 signals in our design failing to meet timing, but in our case they are all related. The report lists violations in order of severity. Usually, you need to look at only the first few failing constraints to get an of idea of where in your design this is failing. Take a look at the first failing signal:

```
Slack:                  -5.748ns (requirement - (data path - clock path skew +...
  Source:               timing/Mmult_M_a_reg_q[7]_M_a_reg_q[7]_MuLt_0_OUT (DSP...
  Destination:          timing/Mmult_M_c_reg_d_submult_01 (DSP)
```

```
Requirement:          20.000ns
Data Path Delay:      25.759ns (Levels of Logic = 3)(Component delays alone ...
Clock Path Skew:      0.046ns (0.696 - 0.650)
Source Clock:         clk_BUFGP rising at 0.000ns
Destination Clock:    clk_BUFGP rising at 20.000ns
Clock Uncertainty:    0.035ns

Clock Uncertainty:          0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
  Total System Jitter (TSJ):  0.070ns
  Total Input Jitter (TIJ):   0.000ns
  Discrete Jitter (DJ):       0.000ns
  Phase Error (PE):           0.000ns

Maximum Data Path at Slow Process Corner: timing/Mmult_M_a_reg_q[7]_M_a_reg_...
  Location             Delay type         Delay(ns)  Physical Resource
                                                     Logical Resource(s)
  -------------------------------------------------  -------------------
  DSP48_X0Y9.M14       Tdspcko_M_B0REG       4.371   timing/Mmult_M_a_reg_q[...
                                                     timing/Mmult_M_a_reg_q[...
  DSP48_X0Y10.B14      net (fanout=2)        1.115   timing/M_a_reg_q[7]_M_a...
  DSP48_X0Y10.M7       Tdspdo_B_M            3.894   timing/Mmult_M_a_reg_q[...
                                                     timing/Mmult_M_a_reg_q[...
  DSP48_X0Y13.A7       net (fanout=1)        1.765   timing/M_a_reg_q[7]_M_a...
  DSP48_X0Y13.P47      Tdspdo_A_P            3.926   timing/Mmult_M_a_reg_q[...
                                                     timing/Mmult_M_a_reg_q[...
  DSP48_X0Y14.C37      net (fanout=18)       1.224   timing/Mmult_M_a_reg_q[...
  DSP48_X0Y14.P1       Tdspdo_C_P            3.141   timing/Mmult_M_a_reg_q[...
                                                     timing/Mmult_M_a_reg_q[...
  DSP48_X0Y12.A1       net (fanout=1)        1.269   timing/M_a_reg_q[7]_M_b...
  DSP48_X0Y12.CLK      Tdspdck_A_PREG        5.054   timing/Mmult_M_c_reg_d_...
                                                     timing/Mmult_M_c_reg_d_...
  -------------------------------------------------  -----------------------...
  Total                                    25.759ns (20.386ns logic, 5.373n...
                                                    (79.1% logic, 20.9% rou...
```

The source and destination fields are the ones you want to look at to figure out what caused this. In our case, `timing/Mmult_M_a_reg_q[7]_M_a_reg_q[7]_MuLt_0_OUT (DSP)` refers to the timing module and the output of one of the multiplications. The destination is the input of another multiplication. The names of your signals at this point in the project building can get pretty cryptic. This is because the tools have already optimized your design and mapped it to FPGA primitives. Usually, enough of the original names are intact that you can get a rough idea of what signals are causing trouble. If you look through the rest of the failed timing constraints, you'll see they are more or less all the same.

So how do we fix this? Well, the root of the problem is that we are trying to do too much in a single clock cycle. We can break our operation into multiple clock cycles by pipelining it. *Pipelining* refers to adding DFFs to break up a single chunk of combinational logic. It will inherently increase latency (it'll now take multiple clock cycles instead of one), but the throughput increases:

```
module timing (
    input clk,    // clock
    input a [8],  // first input
    input b [8],  // second input
    output c [64] // output
  ) {

  .clk(clk) {
    dff a_reg [8];  // register to hold a
    dff b_reg [8];  // register to hold b
    dff c_reg [64]; // register to hold c

    dff temp_a [32];
    dff temp_b [32];
  }

  always {
    a_reg.d = a; // connect inputs
    b_reg.d = b;
    c = c_reg.q; // connect output

    temp_a.d = (a_reg.q * a_reg.q) * (a_reg.q * a_reg.q);
    temp_b.d = (b_reg.q * b_reg.q) * (b_reg.q * b_reg.q);
    c_reg.d = temp_a.q * temp_b.q;
  }
}
```

If you build the project now, all timing constraints will be met. However, note that it now takes three clock cycles for the output `c` to reflect changes on `a` and `b` instead of the previous two cycles.

This design has a best achievable case of 13.078 ns, or a maximum clock frequency of 76.46 MHz. This is nearly twice as good as before and shows an ideal case of pipelining. The latency increased by only a tiny amount, but the throughput almost doubled. Why does adding the extra DFFs nearly double throughput? It is because we broke the block of combinational logic into two sections, allowing each section to operate independently. Before, the second half couldn't start working on a result until the first half computed the intermediate result. Now, the second half is presented with valid data to work with at the start of the clock cycle. As the second half is chugging on some new data, the first half can start working on the next values without worrying about contaminating the intermediate values.

You can continue to add pipeline stages to improve throughput, but this has a limit. As you may recall from "Timing and Metastability" on page 61, DFFs add a delay of their own (the *clock-to-Q* delay). By adding more and more stages, the sum of delays from the DFFs will start to dominate the total delay in your design, and adding more stages can actually make it slower. Also, combinational logic blocks don't have unlimited places to cut them up. In our example, it makes sense to break off the biggest multiplication, as it will take roughly the same time as the smaller multiplica-

tions. But if we wanted to add more states, we would have to split the multiplication operation itself. This can be done (and the `DSP48A1s` can be configured for pipelining), but it becomes much more complicated. If you run into a problem like this, you can usually make a change to your design's overall architecture to make it more efficient rather than trying to optimize an inefficient section.

Pipelining can seem like a magical tool to solve timing problems. Timing failed? Just throw some more DFFs in there. However, this doesn't always work. If your design fails timing because of routing issues, this can be because you are reaching the FPGA's resource limits and the tools are having a hard time finding places to put everything and still efficiently route signals. If this is the case, adding more flip-flops will only add to the congestion and make timing worse. To solve this, make your design smaller/more efficient or get a bigger FPGA.

It is worth mentioning that placement and routing aren't exact processes. The tools have many parameters you can tweak that can cause some designs to fail or meet timing, depending on how they are configured. The Mojo IDE doesn't currently allow you to tweak these, but if you need to, you can open the built project (found in the *work* folder) in PlanAhead and play with stuff there. This isn't usually a good way to try to fix timing, though, as any slight change in your design can break it once again. It's much better to try to find a design solution if possible.

Crossing Clock Domains

Inside your design, all the flip-flops that use the same clock belong to the same *clock domain*. If you can, it is easiest to have just one clock domain in your design. However, some cases require you to have multiple clocks. Usually this happens when you have an external device you want to talk to that throws a lot of high-speed data at you and the data is accompanied by its own clock. In this case, you typically clock part of your design to capture the incoming data, but then you need to transfer that data to your design's primary clock domain.

Before we get into anything complicated, let's look at the simple case of wanting to get a single bit from one clock domain to another. This is a pretty common case, as single bits can be used as flags to signal that a part is ready or busy. Although the later, more complicated techniques will work perfectly fine here, a simple synchronizer can be used. As you may recall from "External Inputs" on page 66, we can use a few flip-flops in a chain to synchronize an unsynchronized signal to our clock. Before, we used these to synchronize external button inputs that have no clock associated with them, but they can also be used to cross clock domains.

Synchronizers work great when you have a single bit and it isn't changing fast. The ideal use case is for flags. Because of metastability issues, you don't know exactly how long it will take for the bit to propagate from one domain to the other. Sometimes it

may take an extra cycle; sometimes it won't. In many cases, this slight unknown delay isn't a big deal.

However, this can cause a problem if you try to use synchronizers for multibit signals. If multiple bits change at the same time, it isn't guaranteed that they will all change at the same time in the other clock domain. For example, if you try to synchronize a 4-bit signal and change from 0 (b0000) to 9 (b1001), you may see 0 (b0000), then 8 (b1000), then 9 (b1001). You could get around this with a delayed `data_valid` flag to signal when to sample the multibit signal, but that is a bit messy, as you'll have to make sure it is delayed enough that you are guaranteed all the bits will be settled. This would cause your bandwidth to be pretty low.

The more sophisticated way to cross clock domains is to use an *asynchronous FIFO*. A *FIFO* is a *first in, first out* buffer. Basically, it takes a stream of values and dispenses them in the order they were received. The *asynchronous* part means that the part that accepts values has a different clock than the part that outputs values. To use one to cross clock domains, you supply both clocks and read and write data from the two domains.

The Mojo IDE has an asynchronous FIFO component available under the Memory category. Take a look at its interface:

```
module async_fifo #(
    SIZE = 4 : SIZE > 0,                        // Size of the data
    DEPTH = 8 : DEPTH == $pow(2,$clog2(DEPTH)) // DEPTH must be a power of 2
  )(
    input wclk,                                 // write clock
    input wrst,                                 // write reset
    input din [SIZE],                           // write data
    input wput,                                 // write flag (1 = write)
    output full,                                // full flag (1 = full)

    input rclk,                                 // read clock
    input rrst,                                 // read reset
    output dout [SIZE],                         // read data
    input rget,                                 // data read flag (1 = get next)
    output empty                                // empty flag (1 = empty)
  ) {
```

The FIFO can be configured to be any `SIZE` (number of bits for each entry) and any `DEPTH` (number of entries) that is a power of 2. To write to the FIFO, you first check that `full` isn't 1. If it is, your write will be ignored. With `full` set to 0, you provide your data on `din` and set `wput` high.

To read from the FIFO, you check that `empty` is 0 (meaning there is data) and then set `rget` to 1 and read the data on `dout`. The FIFO component is what's known as a *first-word fall-through FIFO*. This means that `dout` is valid whenever `empty` is 0. You don't need to wait a clock cycle after setting `rget` to read it.

You'll notice that the FIFO takes two clocks and two reset signals. The reason for both clocks is obvious, but the two resets are a little different. The reason for these is that each reset should be synchronized to the respective clock domain. All the project templates in the Mojo IDE are set up to synchronize the external reset input to the main clock by using the Reset Conditioner component. If you use the FIFO with two clocks, you should use a second Reset Conditioner to synchronize the reset signal to the other clock domain.

So how does the FIFO work? If you take a look at its source, the fundamental block is a dual-port asynchronous RAM. That is RAM that has a read and write port with two clocks. As long as you don't read and write to the same address at the same time, there isn't any ambiguity in what value you will read. The FIFO uses two counters: one to know what address to write data to, and one to know what address to read data from. When you write data, the write address is incremented. If the write address is just below the read address, you know that the FIFO is full. When you read data, the read address increments. If the read address isn't the same as the write address, the FIFO isn't empty.

The tricky part is that the read and write counters live in different clock domains. To cross the clock domains, we use synchronizers. However, the counters are multibit signals! That means we have to worry about invalid intermediate values. We can get around this issue by changing only one bit at a time. If we use simple binary values for the counters, we'll have a problem, because multiple bits will change. When the counter overflows, every bit will change! To overcome this, we use something known as *Gray encoding*. Binary is just an arbitrary way to represent numbers. We could map any combination of 1s and 0s to any number. Gray encoding is an alternative mapping to binary encoding. While binary encoding has convenient math properties, Gray has the special property that any two consecutive numbers differ by only one bit change. Take a look at Table 11-1 for some examples of binary versus Gray.

Table 11-1. Gray encoding

Decimal	Binary	Gray
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

It's easy to convert a binary value to the Gray version. Shift the value to the right by 1 bit, and XOR it with itself: *Gray = (binary >> 1) ^ binary*. Converting back is a little more complicated. The MSB is always the same. From there, each bit is the bit above it XORed with the corresponding Gray bit. For example, a 4-bit Gray number g_3 through g_0 can be converted to a binary number, b_3 through b_0, with the following relations: $b_3 = g_3$, $b_2 = b_3 \wedge g_2$, $b_1 = b_2 \wedge g_1$, $b_0 = b_1 \wedge g_0$.

Using this, and the fact that the values will be incremented by only 1, we can ensure that only a single bit changes, which avoids any invalid intermediate values.

Because of the synchronizers, it will take a few clock cycles from when you write a value to the FIFO for the read side to signal that it isn't empty. You should make sure the FIFO is big enough (`DEPTH` set to a large enough value) so that this won't bottleneck your data. A value of 8 works well if the reading clock is faster than the writing clock (typical case).

If the writing clock of the FIFO is faster than the reading clock, you won't be able to write data every clock cycle because the reading side won't be able to keep up and the FIFO will fill up regardless of its size. You need to make sure that your design can read the data as fast as you plan to write it. If you have bursty write patterns, you need to make sure the FIFO is large enough to absorb the bursts.

FIFOs are useful even when staying in the same clock domain to absorb bursty writes. You could, for example, use a FIFO to absorb writes to the slow serial port. Parts of your design could spit out a bunch of values to write and then forget about it while the FIFO slowly writes them out.

CHAPTER 12

Sound Direction Detection: An Advanced Example

In this chapter, we will use seven microphones, six evenly spaced around a circle with one in the middle, to capture an audio sample and calculate the different directions the sound comes from. In this project, we will push the Mojo's FPGA fairly close to its limits in terms of design size. This will serve as an example for some digital signal processing (DSP), using an FSM to save resources, and a handful of other tricks that you may find useful in your own designs.

Unlike previous chapters, this chapter serves as more of an explanation of the full project rather than a step-by-step walk-through of designing it yourself.

This project also requires specialized hardware, the Microphone Shield, shown in Figure 12-1. This shield has recently been released, and is available online.

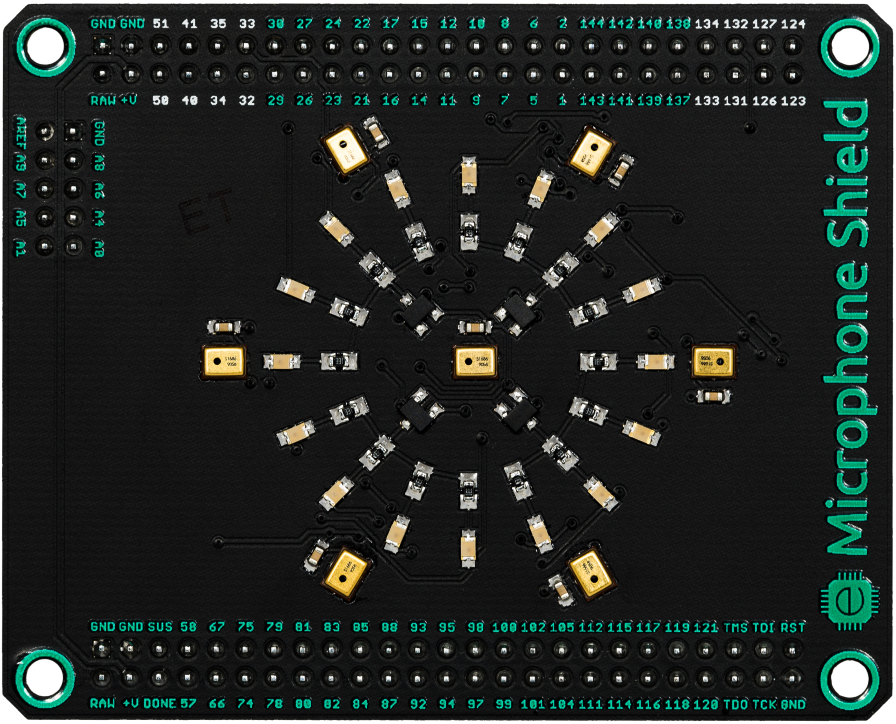

Figure 12-1. Microphone Shield

Theory of Operation

The idea is that we will record a short sample of audio from all seven microphones at exactly the same time. We can then use this to calculate the delay between the microphones and use that to detect the direction of the sound.

We need to make a few assumptions about the sound the shield will be detecting. The first is that all sound comes from the sides and not from above or below. This is required because we have only a 2D grid of microphones. Second, the sound waves have a straight wavefront. This one isn't true because most sound will originate from a single point and have a circular wavefront, but for sources any reasonable distance from our microphone array, it will be a reasonable approximation (a curved line when zoomed in enough looks straight). Finally, we assume that each frequency in a sample comes from a single direction.

So how do we go about calculating the delays between the microphones? If we had a simple pulse and everything else was quiet, it would be quite easy by just looking at the peak of the pulse in each sample. However, the real world is hardly that nice.

Instead, we will be using the *phase* of the different frequencies in each sample. This has two major benefits: the phase is pretty easy to calculate with an FPGA by using a fast fourier transform (FFT), and the other being that we can detect multiple sound sources as long as their frequency components are different enough. Imagine a bird chirping and someone talking. The bird chirps will be substantially higher pitch than the person talking, and we should be able to detect these simultaneously.

If you aren't familiar with FFTs, don't worry too much. All you need to know for this example is that a Fourier transform takes a signal in the time domain, meaning the x-axis of the sample is time, and converts it into the frequency domain. It tells you what frequency sine waves (and their magnitudes) you would have to add together to get the exact same signal back. If you've ever seen a music equalizer, you've seen an FFT in action. This is exactly what the demo of the Clock/Visualizer Shield does.

So after collecting a short sample from all seven microphones, we can run each one through an FFT to get the frequency components. The FFT output for each frequency is a complex number. The real portion of the number corresponds to the magnitude of the sine portion, and the imaginary portion corresponds to the cosine portion. By adding together sine and cosine waves of varying amplitudes, you can create a sine wave with any phase.

The raw output of the FFT isn't particularly useful to us. Instead, it would be much better if we knew the phase and magnitude of each frequency. To do this, we need to convert the complex number, which can be thought of as converting a Cartesian coordinate (if that helps you) into a polar coordinate. Basically, if we were to plot the complex number on a regular 2D space, instead of the x and y position of the coordinate, we want to know the angle and distance it is from the origin. Again, this isn't too bad to do with an FPGA, as you'll see later.

With the phase of each frequency calculated, we can subtract the phase of the center microphone from each of the surrounding six microphones to get a phase offset for each one. Using the formula *delay = phase offset / frequency*, we could calculate the delay for that frequency. However, scaling the delay by a constant factor (the frequency) for all six microphones won't make a difference later, and we can use this fact to avoid a costly division in the FPGA. Instead, we will simply use the phase offsets as if they were the delays, since they are proportional to them.

Now that we have a delay for each microphone relative to the center microphone, we need to combine these to get an overall direction. To do this, we need to scale each of the microphone's location vectors by their corresponding delay and sum them. This will give us a single vector pointing in the direction of the sound source.

Figure 12-2 shows this geometrically. I drew in only three microphones for simplicity. Adding the other three would make the sum of the scaled vectors twice as long but wouldn't change the direction due to symmetry. I also drew this so that the sound is coming from the *y* direction and the microphones are rotated by Φ instead of the sound coming in at angle Φ. This will make it a little easier to show that this method works later. The black circles represent the locations of the microphones, and their coordinates are labeled.

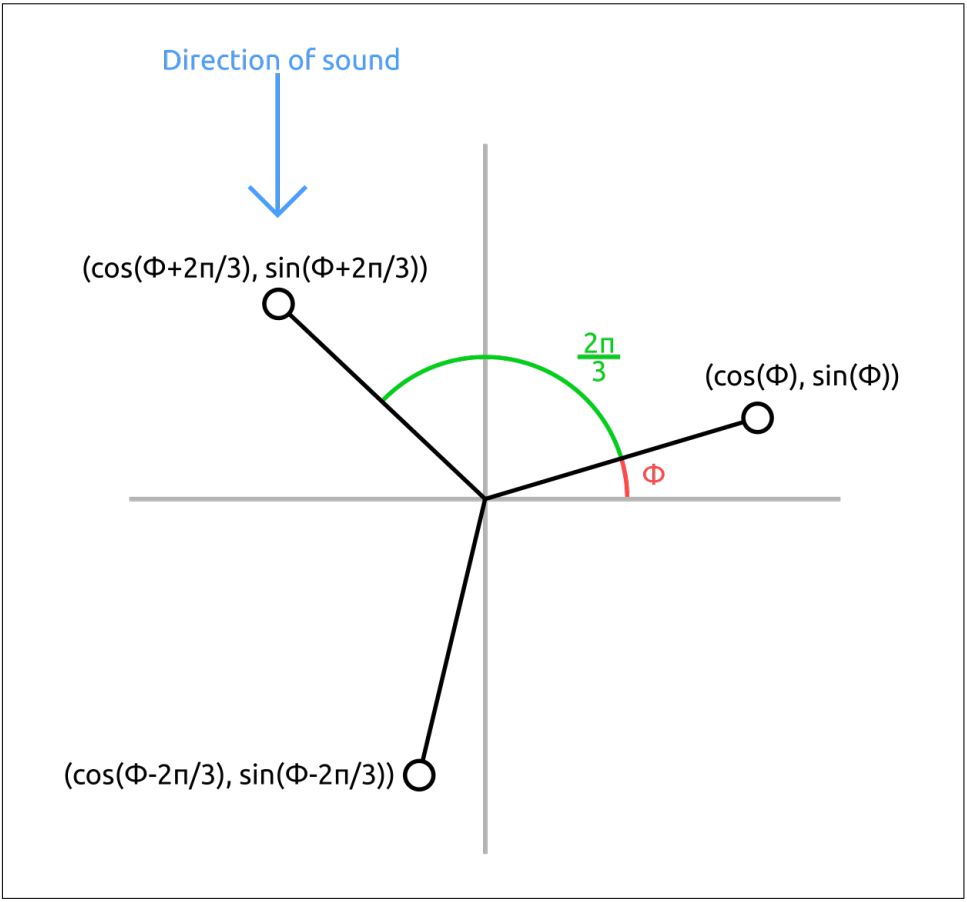

Figure 12-2. Microphone array

The delay of each microphone to the origin (center microphone) is proportional to the *y* value of the microphone's location. We can draw these in, as shown in Figure 12-3.

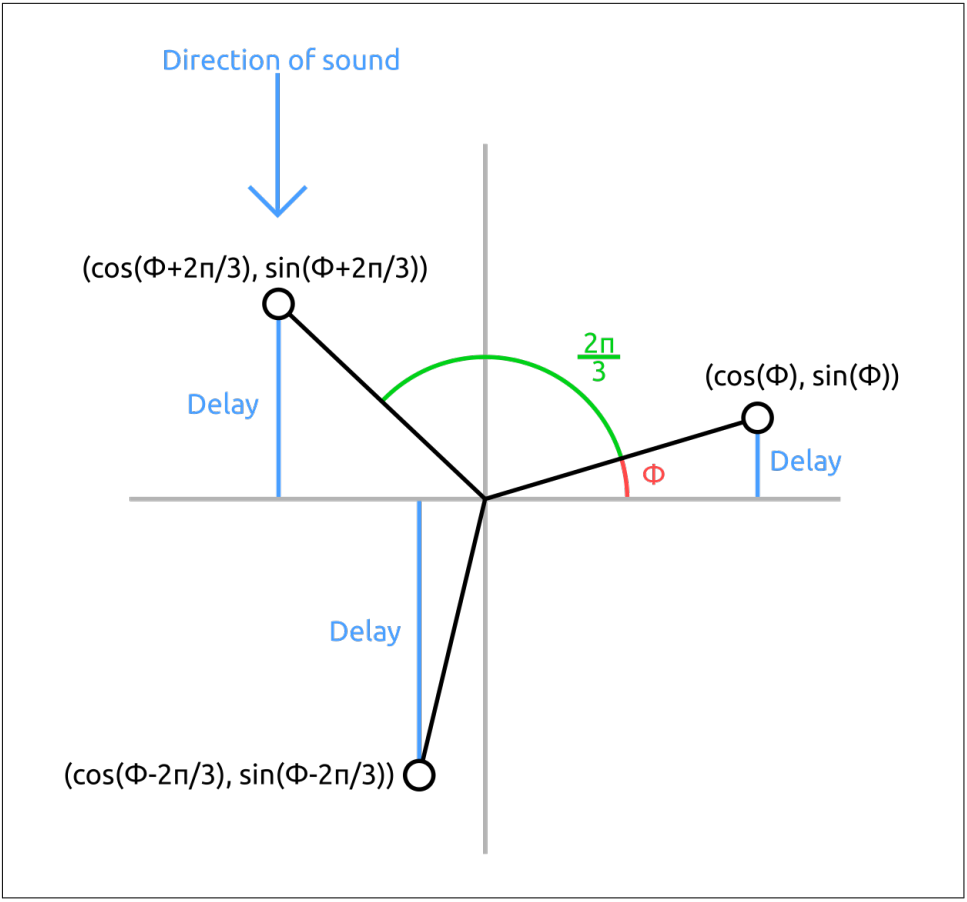

Figure 12-3. Microphone array with delays

If we take the location of each microphone and scale it by the corresponding delay, we get the new purple lines shown in Figure 12-4. Note that the bottom microphone has a negative delay, so the vector points in the opposite direction.

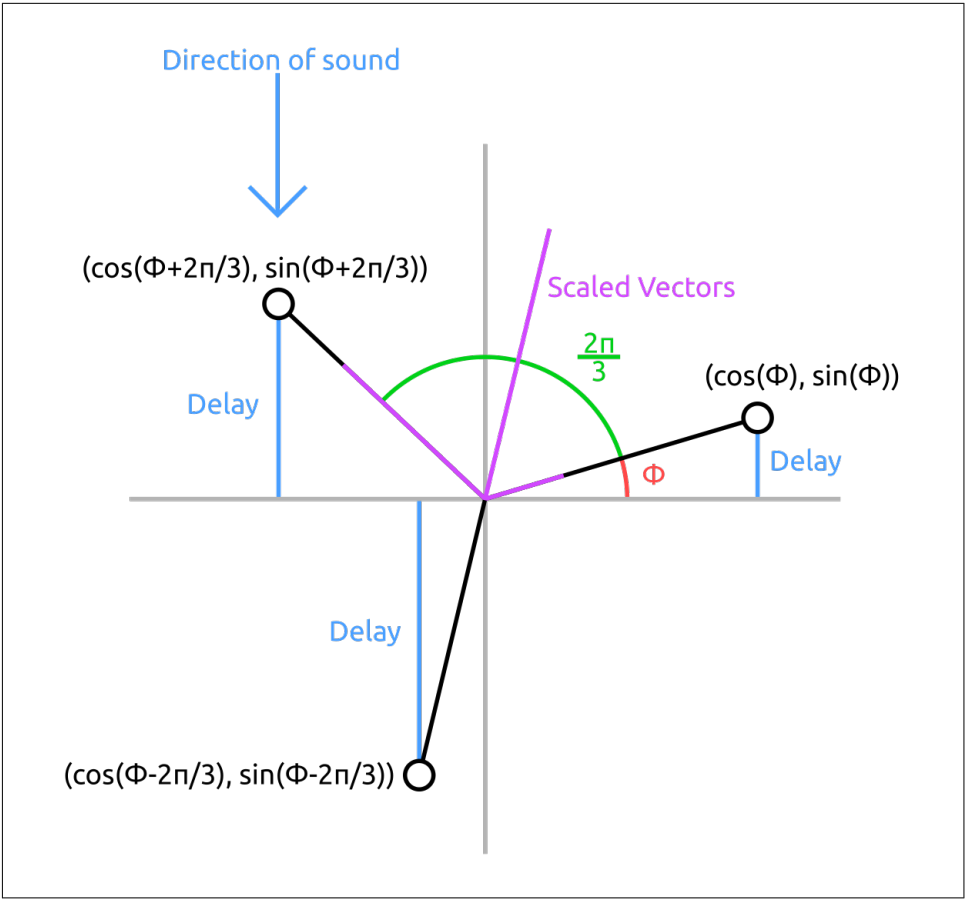

Figure 12-4. Microphone array with scaled vectors

Finally, we can take the three scaled vectors and sum them together by moving them tip-to-tail. This is shown by the light purple lines in Figure 12-5. The orange vector is the result of the summation of the three scaled vectors.

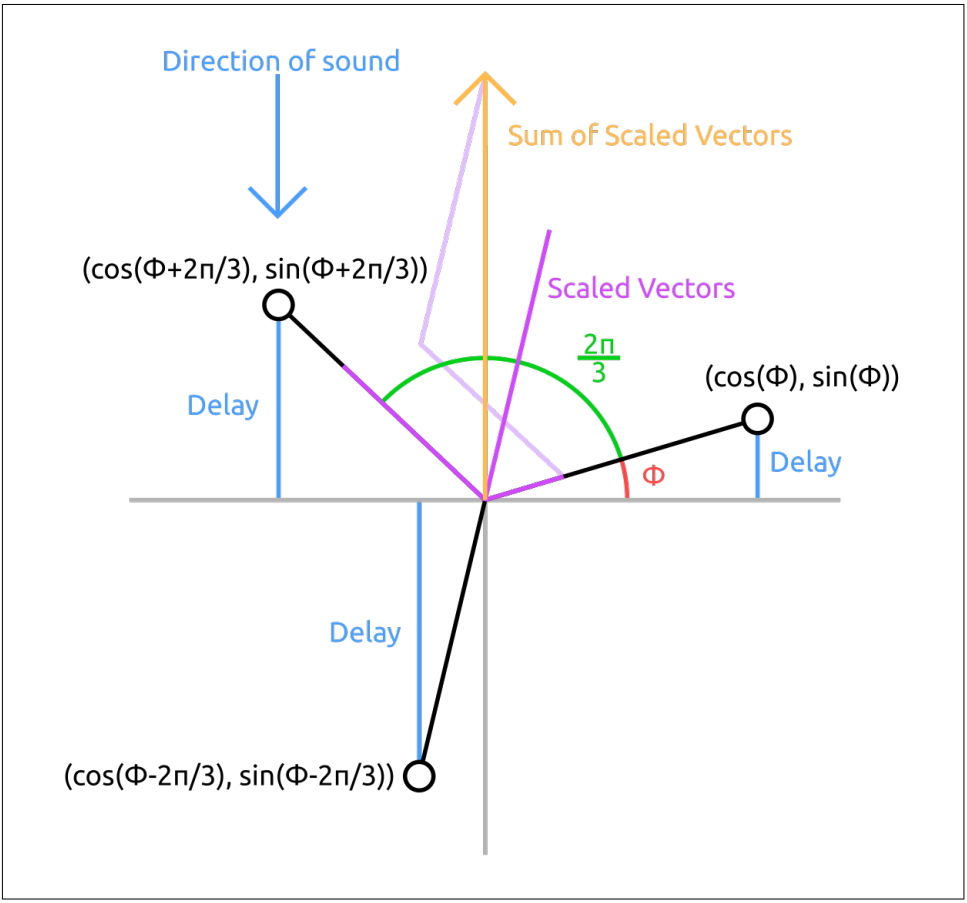

Figure 12-5. Microphone array

Notice that the x components of the summed scaled vectors cancel, so the resulting sum points only in the positive y direction. To prove that this method works, we need to show that the x components cancel and the y components sum to a positive value for any value of ϕ. We don't care about the magnitude of the resulting vector, only the direction.

You can find a full proof at "Direction Detection Proof" on page 211.

Now that we have the angle for the frequency, we need to aggregate each angle into something useful. I chose to bin them into 16 directions so that it would be easier to use and display them on the LEDs. I used the magnitude from the center microphone to weight the importance of each frequency's contribution. This was done by iterating through each frequency, determining the bin it belonged to, and keeping a running total of the magnitude of each bin.

The final output is the 16 values representing how much noise came from that direction.

Implementation Overview

Now that we have an idea of how this is going to work, we need to come up with a plan for implementing this in hardware.

First, we need to gather audio samples from all seven microphones at exactly the same time. The microphones on this shield are *pulse-density modulation* (PDM) microphones, meaning that they provide a series of 1-bit pulses at a high rate (2.5 MHz in this case) that we can pass through a low-pass filter (basically, a moving average) to recover the audio signal. We also decimate the signal by a factor of 50, so our sampling rate becomes 50 KHz.

With the seven audio samples captured, we need to feed each of them through an FFT to extract their frequency information. The output of the FFT is complex numbers, but we need it to be in phase-magnitude form, so we then pass these values through a module that calculates the new values.

With the phase-magnitude representation of all the samples, we can then subtract the phases of the six surrounding microphones from the center one to get the delays. We need to be careful here because after the subtraction, the phase difference can be outside the +/– pi range. If it is, we need to add or subtract 2 pi to get it back into range.

The calculated phase differences are equal to the delay multiplied by the frequency. Because we are working with one frequency at a time, it is really just the delay scaled by a constant. We can use this fact to avoid having to divide by the frequency.

We then scale the six microphone location vectors by the corresponding phase differences (delays) and sum their components. This gives us a vector that points in the direction of the sound source for this frequency. However, we care only about the direction of this vector, as the magnitude is pretty meaningless. We can convert the Cartesian vector into a phase-magnitude representation by using the same module as before to extract the phase (angle).

Repeating this process for all the frequencies gives us an angle for the direction of sound for each frequency. We can pair each of these directions with the magnitude (volume) of that frequency from the center microphone to find out how relevant it is.

This in itself could be the output of our design, but it is a little more useful to bin the directions into a handful of angles. In our case, we will assign each into one of 16 equally spaced bins. All the magnitudes of the frequencies that fall into a bin are summed to get that bin's overall magnitude. These 16 sums are the final output and represent the amount of sound that came from each bin's direction.

We could implement this design as a full pipeline with each stage simply feeding into the next. Figure 12-6 shows what that could look like.

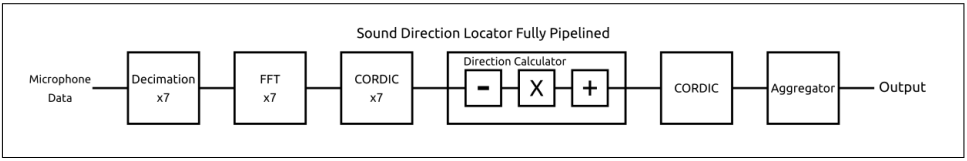

Figure 12-6. Sound direction locator fully pipelined

This design would have the highest throughput, but it would also take up a lot of resources. In fact, it would take up way more than we have available in the Mojo's FPGA. However, we can instead perform each step in sequence and take advantage of the fact that a lot of the steps require the same operations, just on different data. In a full pipeline we would need seven FFTs and eight CORDICs (the Cartesian-to-phase-magnitude converts). However, we can reuse just one of each and save a ton of resources.

Figure 12-7 is a drawing of the *data-path* of the circuit. The data-path shows the way data flows through a design, but it does not, for simplicity, show the control logic that controls the multiplexers and other flow decisions. The fully pipelined version doesn't need any control logic because the data just flows from one end to the other. However, we will need to create an FSM to control the compact version. The steps the FSM will need to take are outlined in the following paragraphs.

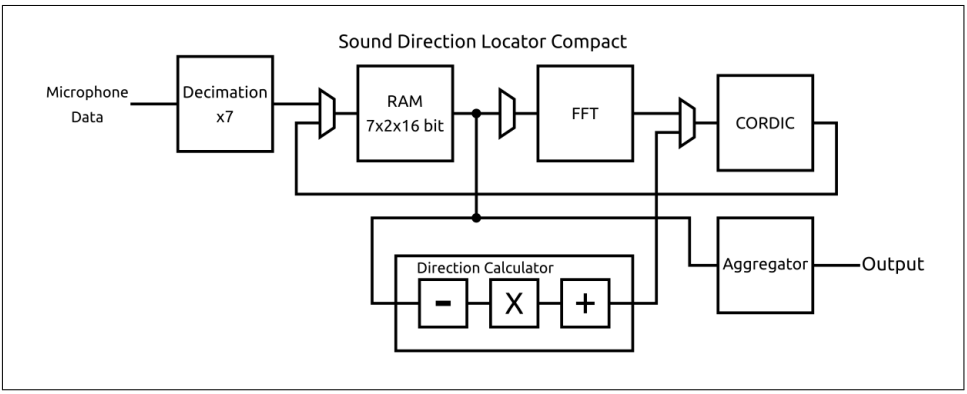

Figure 12-7. Sound direction locator compact

First, samples from the microphones pass through the decimation filter and are stored in the RAM. The RAM is organized into seven groups of two (14 RAMs total), each 16 bits wide. The seven groups correspond to the seven microphones, and two RAMs in each group will store even and odd samples, respectively. The reason for the 7 x 2 arrangement will become clear later.

When the blocks of RAM are full of sample data, the data is passed one channel at a time to the FFT. The data is also passed through a Hanning window (not shown) to minimize leakage in the FFT. The purpose of this is outside the scope of this book, but it comes down to multiplying the samples by the Hann function that is stored in a ROM. The output of the FFT is fed into the CORDIC to convert it to phase-magnitude format and then written back into the RAM, overwriting the original channel's sample data. When writing back to the RAM, the seven groups still correspond to each channel, but the two values in each group now are for the phase and magnitude values instead of even and odd samples. By having these two values in different RAMs, we can easily read or write them simultaneously. This is repeated seven times, once for each channel. After this step is done, the RAM contains phase and magnitude data for each channel.

Because the sample data we are feeding into the FFT is all real (no imaginary components), the output of the FFT will be symmetrical. This means that even though each frequency has two values related to it, we have half the number of frequencies as we did samples, so we have the exact same number of values to store in the RAM.

The next step is to take the phase-magnitude data for each frequency and pass it through the direction calculator to get the directional vector for that frequency. The angle (phase) of that vector is then extracted using the same CORDIC as before. The output is then saved back into the RAM. This time, we write the data to group 0, as we don't need the phase-magnitude data for this microphone (nor microphones 1–5) anymore.

Finally, we feed the phase data from the last step and the magnitude data from microphone 6 (the center microphone) into the aggregator. The aggregator adds each sample to its corresponding bin and outputs the final results.

Even with all this reuse, this design still use 77% of the LUTs, 32% of the DFFs, and occupies 94% of the slices in the Mojo. It just fits!

Implementation

Now that we have a road map of what we need to design, let's get into the code. You can find the full source in the GitHub repo for this book.

We will start with the microphones and work our way through the steps of the sound locator.

PDM Microphones

This project relies on PDM microphones. These are a common type of microphone and easy to interface with an FPGA because they have a digital output. As noted previously, *PDM* stands for *pulse-density modulation*, which means that the density of

pulses is correlated to the pressure on the microphone. Because of this simple interface, we need a lot of pulses to be able to get any real definition of the underlying signal. The microphones on the Microphone Shield can output between 1 and 3.25 million pulses per second, depending on the clock provided to it. In our design, we will use the nice middle value of 2.5 million per second.

To convert this high-frequency, low-resolution pulse train into a more useful lower-frequency, higher-resolution signal, we will use a *cascaded integrator–comb* (CIC) filter. This type of filter is useful for changing sampling rates, decimation (decrease), or interpolation (increase). Lucky for us, Xilinx's CoreGen tool can be used to generate the filter.

If you have the full Sound Locator project open, you can launch CoreGen and take a look at the `decimation_filter` core. It was created from the CIC Compiler version 3.0, which can be found under Digital Signal Processing → Filters → CIC Compiler and is shown in Figure 12-8.

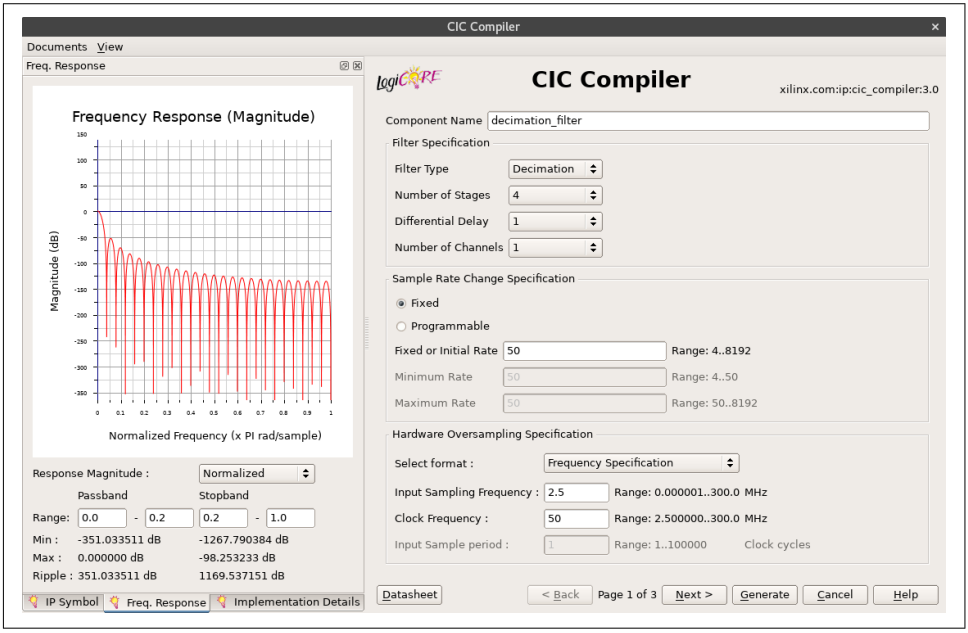

Figure 12-8. CIC Compiler page 1

Here we specify that it is a decimation type filter and should have a decimation rate of 50. This will convert the 2.5 MHz input into 50 KHz output. Take a look at the frequency response chart, which shows that it is a low-pass filter. You can play with the other parameters to get it to attenuate the higher frequencies more at the cost of more hardware. For our use, it doesn't really make a difference.

If you look at the second page, shown in Figure 12-9, it shows the filter is capable of outputting 25 bits per sample. However, this is set to 20, so the last 5 bits are truncated. This was found empirically to be a good value for sensitivity, and we will be using only 16 bits for each sample anyway. The extra MSBs will be used to check for overflow but will otherwise be ignored.

We also have this set not to use *Xtreme DSP Slices*, as we don't have enough to spare in the FPGA. This option will use the built-in multipliers when selected instead of using the general fabric of the FPGA. However, it will use two multipliers per filter, and we have seven filters. The FPGA on the Mojo has 16 multipliers, so 14 just for this stage would be way too much.

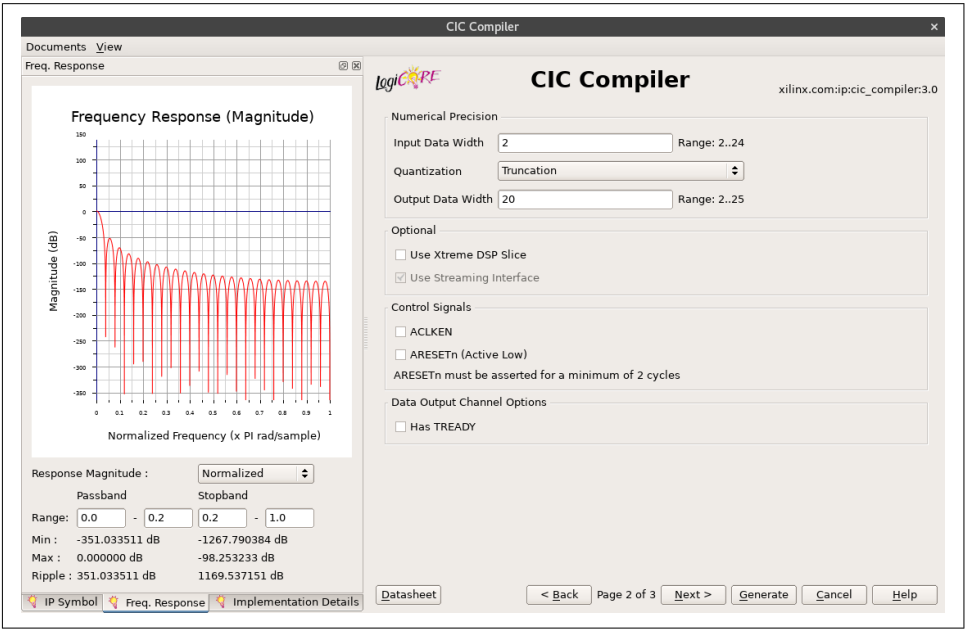

Figure 12-9. CIC Compiler page 2

For more information on the CIC filter, check out the Xilinx LogiCORE documentation.

Armed with the filter, we can now look at the *pdm_mics.luc* file to see how it is used.

In this module, we need to generate a clock for the microphones. This is a 2.5 MHz signal that is 1/20th of the 50 MHz system clock. To do this, we can use a counter that counts from 0–9 (10 cycles) and toggle the clock each time it overflows. We can detect overflows when the MSB falls.

On each rising edge of the microphone clock, we have another bit of PDM data from each microphone. We first need to convert the single-bit value of 0 or 1 into a 2-bit signed value of –1 or 1 for the CIC filters.

All that's left is to feed the data into the CIC filters and output the data from them after converting it to 16 bits. The CIC filter data is signed, so when we convert to 16 bits and want to saturate on overflows, we need to check for negative and positive overflow separately:

```
module pdm_mics (
    input clk,             // clock
    input rst,             // reset
    output mic_clk,        // clock for all the microphones
    input mic_data [7],    // data from each microphone
    output sample [7][16], // sample from all 7 microphones
    output new_sample      // new sample flag
  ) {

  .clk(clk) {
    .rst(rst) {
      counter clk_ctr (#SIZE(4), #TOP(9));       // clock divider counter
      dff mic_clk_reg;                           // mic_clk dff
    }
    edge_detector clk_edge (#RISE(0), #FALL(1)); // clock counter reset detector
    edge_detector new_data (#RISE(1), #FALL(0)); // clock rising edge detector
  }

  // decimates by a factor of 50
  decimation_filter dfilter [7] (.aclk(clk));

  const SAMPLE_MSB = 19;
  const SAMPLE_LSB = 0;

  // used to store unused MSBs
  sig left_over [SAMPLE_MSB - SAMPLE_LSB + 1 - 16];

  var i;

  always {
    // generate a clock at 1/20 the system clock (2.5 MHz)
    clk_edge.in = clk_ctr.value[3];   // this bit will fall when clk_ctr resets
    if (clk_edge.out)                 // if fall was detected
      mic_clk_reg.d = ~mic_clk_reg.q; // toggle the mic clock

    new_data.in = mic_clk_reg.q;      // detect rising edges = new data

    mic_clk = mic_clk_reg.q;          // output mic clock

    // data valid at rising edge of mic clock
    dfilter.s_axis_data_tvalid = 7x{new_data.out};

    // all decimators are identical so we can use any tvalid flag for new_sample
```

```
      new_sample = dfilter.m_axis_data_tvalid[0];

      // for each mic
      for (i = 0; i < 7; i++) {
        // convert 0 or 1 into -1 or 1
        dfilter.s_axis_data_tdata[i] = mic_data[i] ? 8d1 : -8d1;
        sample[i] = dfilter.m_axis_data_tdata[i][SAMPLE_LSB+:16];
        left_over = dfilter.m_axis_data_tdata[i][SAMPLE_MSB:SAMPLE_LSB+16];

        // check for overflow and saturate
        if (!left_over[left_over.WIDTH-1] && (|left_over))
          sample[i] = 16h7fff;
        else if (left_over[left_over.WIDTH-1] && !(&left_over))
          sample[i] = 16h8000;
      }
    }
  }
```

FFT

Before we jump into the `sound_locator` module, let's take a look at the two other cores we need from CoreGen: the FFT and CORDIC cores. Again, with the Sound Locator project open, fire up CoreGen from the Mojo IDE and take a look at the `xfft_v8_0` core.

The first page of the FFT wizard, shown in Figure 12-10, allows you to choose the number of channels (the number of FFTs you want to compute at once), the transform length (number of samples), the system clock, and the architecture. We will be using 512 samples per iteration, which seems to be a nice balance between latency and accuracy. The FFT architecture is set to Radix-2 Lite, Burst I/O, which will result in the smallest implementation at the cost of speed. We don't care that much about speed, but resources are at a premium. Even at the slowest architecture type, it will take only 5,671 cycles to compute the transform. With our 50 MHz clock, that is a small 113.4 µs (about 1/10,000th a second). The fastest architecture is about five times faster, but takes a lot more resources that we don't have. We could likely use the next size up for a modest speed improvement, but, again it won't make an appreciable difference for this use. It takes 100 times longer for us to capture the sample in the first place.

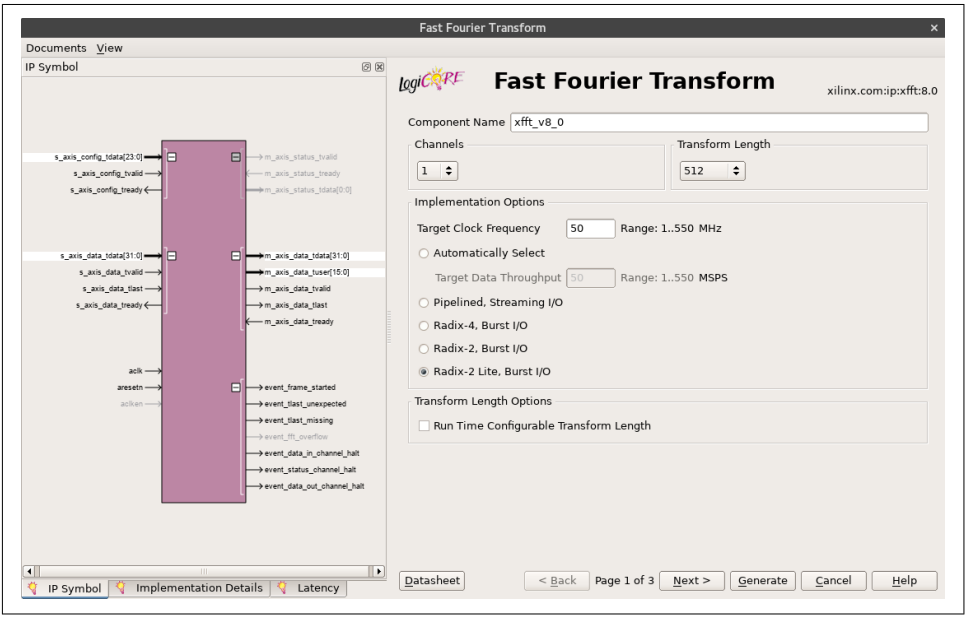

Figure 12-10. FFT Wizard page 1

On the second page, shown in Figure 12-11, we specify more details about the internals of the FFT. We are using fixed-point data, so the data format is set to fixed point. The samples from the microphones are 16 bit, so that's what the input data width is set to. The phase-factor width is the size of some values stored in a ROM that are used when computing the FFT. Higher values will result in a slightly more accurate result. The accuracy for our purposes isn't too important, as 16 bits is more than enough.

Scaling is used to save resources. Without scaling, the values continue to grow in the different FFT steps that require a much wider data path. However, if you enable scaling, you need to provide a scaling schedule that tells the FFT core when to truncate data. This is set at runtime, and I found one that worked well for the microphone data using Xilinx's Matlab module that simulates the FFT.

The rounding mode option is another trade-off of accuracy and resources. Truncation is basically free, while convergent rounding has a small cost.

The control signals are fairly self-explanatory; you can enable a reset and a clock enable if you need them. We need only the reset.

An FFT will naturally produce values in bit-reversed order. For an FFT of eight samples, the output order would be 0, 4, 2, 6, 1, 5, 3, 7. In binary, this is 000, 100, 010, 110, 001, 101, 011, 111. If you reverse the bits around, this becomes 0, 1, 2, 3, 4, 5, 6, 7. If you can work with the output in this order, the core can load and unload data

simultaneously. Otherwise, loading and unloading must take place in different stages. By enabling XK_INDEX in the Optional Output Fields section, we get the index of each output value, so the order doesn't really matter for us.

Finally, the throttle scheme is to specify whether whatever is receiving the output can stop the core from outputting data. In other words, will the hardware using the output ever not be able to accept data? If it can't for any reason, you need to use Non Real Time. This enables the `m_axis_data_tready` flag to signal that data can be output. If you set this to Real Time, it will spit out data as soon as it is ready, no matter what. In our case, we are feeding the data into the CORDIC core, which can be busy, so this needs to be Non Real Time.

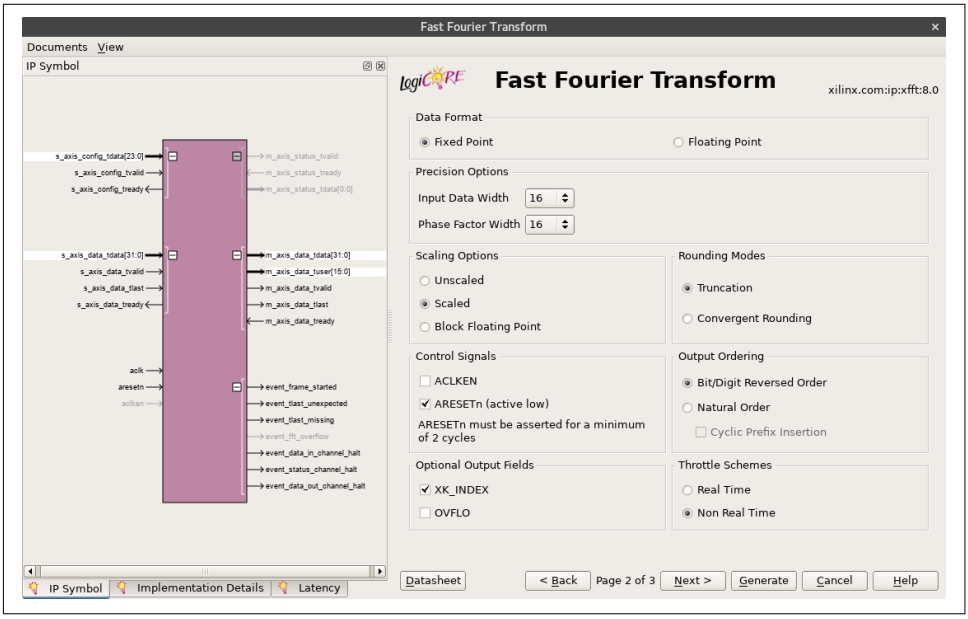

Figure 12-11. FFT Wizard page 2

The final page of the wizard, shown in Figure 12-12, is all about hardware usage. The first section, Memory Options, allows you to choose whether you want to use block RAM or distributed RAM. We have plenty of block RAM left over in our design, so it only makes sense to take advantage of it.

In the Optimize Options section, choose to use the DSP multipliers. The FPGA on the Mojo has 16 of these, which should be used when you can. For the CIC filter, we set it not to use these because they would use two each, and we have seven filters, so it would take a total of 14 multipliers. That is too much (our design only has eight extra). In this case, the FFT will use only three, and we can spare them. You can see

how many it will use under the Implementation Details tab on the left side of the wizard.

In general, if you have special resources available (block RAM and DSPs, in this case), use them. They will generally make your design faster and smaller, and if you don't take advantage of them, they will still be sitting in your FPGA.

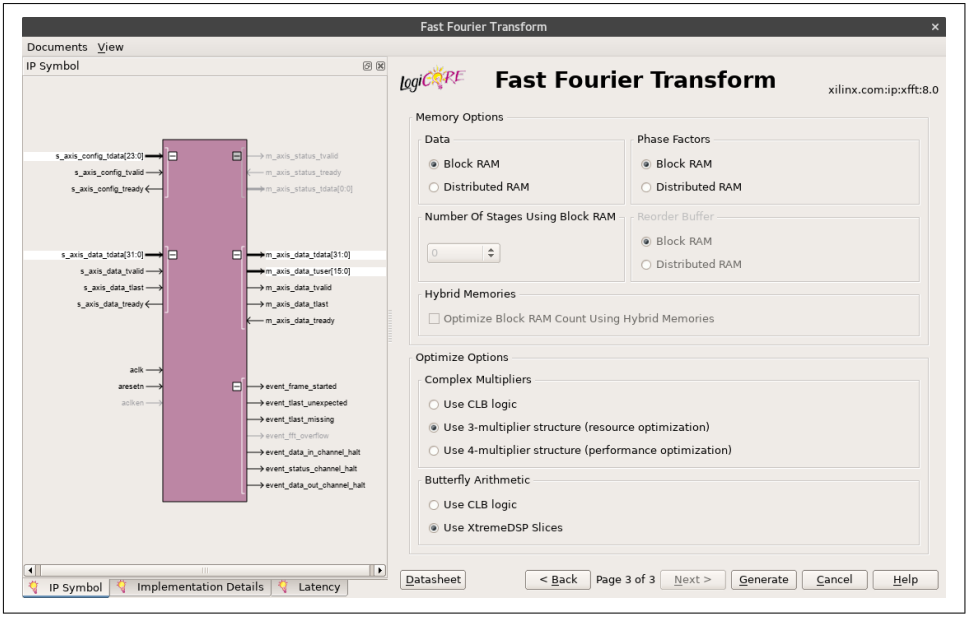

Figure 12-12. FFT Wizard page 3

For more information on the FFT core, check out the Xilinx LogiCORE documentation.

CORDIC

Coordinate Rotation Digital Computer (CORDIC) is an algorithm for efficiently calculating hyperbolic and trigonometric functions. The algorithm is capable of rotating vectors that can be cleverly used to convert to/from Cartesian and polar (magnitude and angle) notations, compute sin and cos, compute sinh and cosh, compute arc tan, compute arc tanh, or even compute square roots. We will be using it to convert from Cartesian to polar coordinates.

Open the `mag_phase_calculator` core in CoreGen.

On the first page of the wizard, as shown in Figure 12-13, we have options to specify the mode of operation as well as accuracy, latency, and area trade-offs.

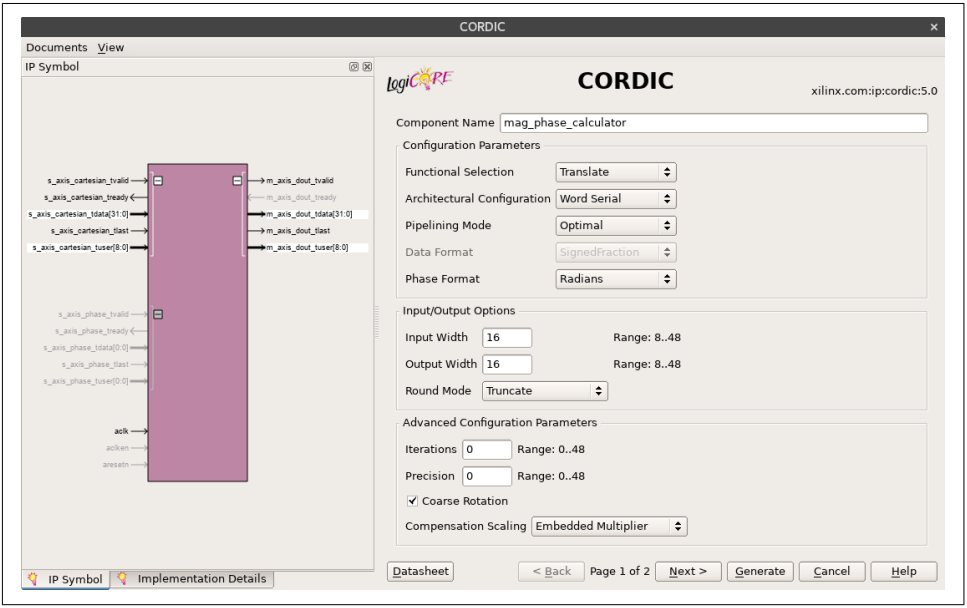

Figure 12-13. CORDIC Wizard page 1

The Functional Selection option selects the mode of operation. In our case, Translate is selected for Cartesian-to-polar conversion.

The Architectural Configuration option gives an area versus latency trade-off. The Parallel option allows the core to spit out a new value every clock cycle by replicating a bunch of hardware. The *Word Serial* option reuses hardware but can work on only one value at a time. Optimizing for area, the Word Serial mode was selected.

The Pipelining Mode is a performance (maximum clock speed) versus area and latency trade-off. The Optimal option will pipeline as much as possible without using extra LUTs. The Maximum option will pipeline after every stage.

The Phase Format option is important, as it dictates the output format. Either option will output a fixed-point value with the three MSBs being the integer portion. For example, 01100000 would be 3.0, and 00010000 would be 0.5. In the Radians case, this value is the angle in radians. For Scaled Radians, this value is the angle divided by pi. We are using radians for simplicity.

The Inout/Output Options section is pretty self-explanatory. We are using 16-bit inputs and outputs, and truncation for internal rounding as it is the cheapest option.

Under Advanced Configuration Parameters, leaving Iterations and Precision at 0 will cause the wizard to automatically set these based on the output width. The Coarse Rotation option allows us to use the full circle instead of just the first quadrant. Compensation Scaling is used to compensate for a side effect of the CORDIC algorithm.

By enabling this, the core will output unscaled correct values at the expense of some resources. You can select the resources to use in the drop-down. In our case, we are using Embedded Multiplier, as we still have a few `DSP48A1s` to use. We could have used the BRAM option as well, as we have plenty of that too.

On the second page of the wizard, as shown in Figure 12-14 you can configure the input and output streams. The TLAST and TUSER options give you extra inputs to the module that will output the values seen when the corresponding inputs have been processed. It's a way to pass along data and keep it in sync. In our case, we will need the address of the values and where the last sample is, so both are enabled. The address size is 9 bits, so we set TUSER's width to 9.

The Flow Control option will enable buffers on the input when Blocking is specified. This is more useful when you have the CORDIC in a different mode when both input streams are used, as it will force them to be in sync. The NonBlocking option uses less resources, and there is no real benefit to Blocking for us.

Finally, we don't care about a reset, as the CORDIC will have flushed itself by the time the FSM ever reaches it upon a reset.

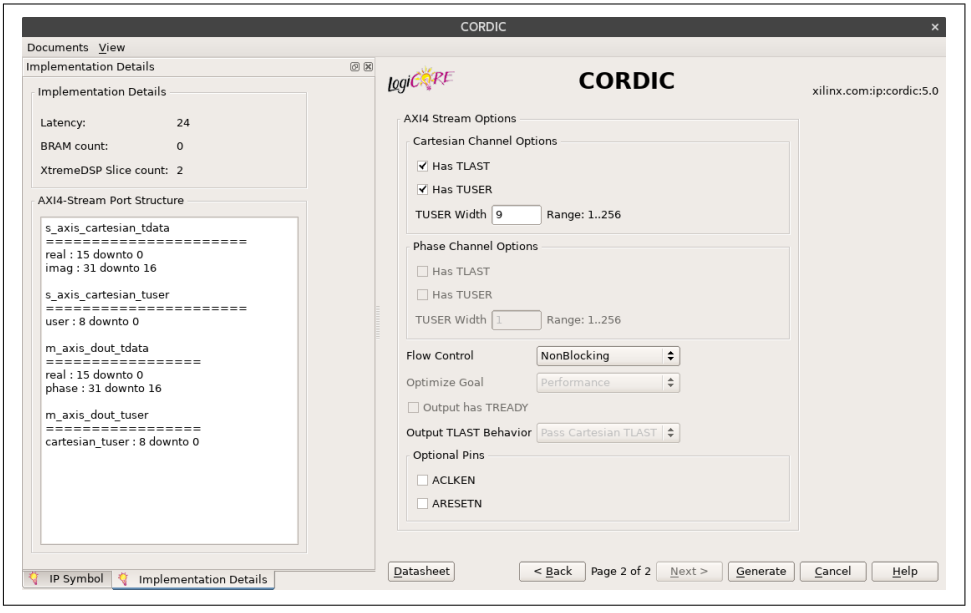

Figure 12-14. CORDIC Wizard page 2

For more information on the CORDIC core, check out the Xilinx LogiCORE documentation.

Now that we have all the pieces, let's dig into the `sound_locator` module itself. The module is too big to replicate here in its entirety, but we will dissect the states of the

FSM. The FSM has four main stages: CAPTURE, FFT, DIFFERENCE, and AGGREGATE. The CAPTURE state is responsible for storing the microphone samples into RAM. In FFT, these samples are converted into their frequency representation. The DIFFERENCE state takes this information and calculates a direction for each frequency by using the difference in phase for each microphone. Finally, AGGREGATE combines all the data together into the 16 directional bins and outputs the final result.

CAPTURE State

The CAPTURE state is pretty straightforward. We simply wait for new samples to come in from the microphones and write them to RAM, incrementing our address counter until the RAM has been filled.

There is a little fancy notation for assigning the address value to all 14 RAM modules because they are in a 7 x 2 array. The first line in this state duplicates addr.q (excluding the LSB) into the 7 x 2 x 8 array.

The LSB of addr is used to select between even and odd RAMs. This little quirk is because we want the two RAMs separate for when we store magnitude and phase data in them later. We assign both even and odd RAMs the same data, but enable the write only on the corresponding one.

Finally, after the RAM is full, we perform some initialization for the next state and move on to FFT:

```
state.CAPTURE:
  // set each channel's write address to addr.q (minus the LSB)
  ram.waddr = 7x{{2x{{addr.q[addr.WIDTH-1:1]}}}};

  if (new_sample_in) {
    // write each sample to RAM
    for (i = 0; i < 7; i++) {
      ram.write_data[i] = 2x{{sample_in[i]}};
      // write alternating samples to the upper and lower channels
      ram.write_en[i][addr.q[0]] = 1;
    }

    addr.d = addr.q + 1;
    // if all samples captured move on to next stage
    if (addr.q == SAMPLES - 1) {
      addr.d = 0;
      load_ch_ctr.d = 0;
      unload_ch_ctr.d = 0;
      wait_ram.d = 1;
      state.d = state.FFT;
    }
  }
```

FFT State

In this stage of the calculation, we have two processes going on. The first is responsible for keeping the FFT feed, and the second is for writing the result from the CORDIC into RAM. The FFT feeds directly into the CORDIC.

The feeding process requires a little finesse, because reading from the RAM takes a clock cycle. The flag `wait_ram` is used to ensure that the RAM is outputting the value for address 0 when we start. If at any point the FFT can't accept more data but we are feeding it data, we'll need to save the value that is coming out of the RAM because on the next cycle it will be gone. When resuming feeding data into the FFT, we then feed it the saved value before resuming reading from the RAM.

Before feeding the FFT, we also pass the data through a Hanning window. The Hann ROM has a single-cycle latency like the RAM, so we also need to save its value if the FFT can't accept values. The Hann value and microphone sample are multiplied together. The Hann value is a 1.15 fixed-point number (1 integer bit, 15 decimal bits), so the result of the multiplication is shifted 15 bits to the right before being passed to the FFT. The idea is the same as if you were multiplying two decimal numbers. For example, 2 x 1.3 can be looked at as (2 x 13) / 10. Note that the multiplication should be a signed multiplication, so both operands are wrapped by the `$signed` function to ensure this.

Once we've filled the FFT, we increment the channel and wait for the FFT to be ready to accept more data. After all seven channels have been loaded, we wait for the state to change:

```
state.FFT:
  // read from addr.q minus LSB
  ram.raddr = 7x{{2x{{addr.q[addr.WIDTH-1:1]}}}};

  // only load the seven channels
  if (load_ch_ctr.q < 7) {
    // if we have to wait for the RAM to catch up
    if (wait_ram.q) {
       wait_ram.d = 0;      // reset flag
      addr.d = addr.q + 1; // increment address
    } else {
      // if the fft was ready but now isn't we need to save the
      // output from the RAM for when the FFT is ready
      if (xfft_ready.q && !xfft.s_axis_data_tready) {
        last_value.d = ram.read_data[load_ch_ctr.q][addr.q[0]];
        last_hann.d = hann.value;
      }

      // if the FFT is ready to accept data
      if (xfft.s_axis_data_tready) {
        // if the FFT was ready last cycle use the RAM output directly,
        // otherwise use the saved value
```

```
        sample = xfft_ready.q ?
          ram.read_data[load_ch_ctr.q][addr.q[0]] : last_value.q;
        hann_value = xfft_ready.q ? hann.value : last_hann.q;

        // multiply each sample by the HANN window
        mult_temp = $signed(sample) * $signed(c{1b0,hann_value});
        hann_sample = mult_temp[15+:16];
        // imaginary part of FFT is 0
        xfft.s_axis_data_tdata = c{16b0, hann_sample};
        xfft.s_axis_data_tvalid = 1;
        addr.d = addr.q + 1;

        // addr.q will be 0 if the fft was stalled when waiting for the
        // last sample

        // if we've read all the samples
        if ((addr.q == SAMPLES - 1 && !xfft_ready.q) || addr.q == 0) {
          xfft.s_axis_data_tlast = 1;
          addr.d = 0;
          wait_ram.d = 1; // wait for RAM to read addr 0
          load_ch_ctr.d = load_ch_ctr.q + 1;
        }
      }
    }
  }
```

We need to connect the FFT output to the CORDIC to get the magnitude-angle representation. The CORDIC conveniently lets us pass the address and last flag through it so that it stays in sync with the other data. The `tready` and `tvalid` handshaking flags ensure that data is transferred only when there is data and the CORDIC can accept it:

```
xfft.m_axis_data_tready = mag_phase.s_axis_cartesian_tready;
mag_phase.s_axis_cartesian_tdata = xfft.m_axis_data_tdata;
mag_phase.s_axis_cartesian_tvalid = xfft.m_axis_data_tvalid;

// pass the address info through the user channel so it is available
// when the mag_phase data has been processed
mag_phase.s_axis_cartesian_tuser = xfft.m_axis_data_tuser[addr.WIDTH-1:0];
mag_phase.s_axis_cartesian_tlast = xfft.m_axis_data_tlast;
```

Finally, we need to unload the CORDIC data into the RAM. We have the address to write from the `tuser` field, and we keep track of the channel number by counting the `tlast` flags.

After we have unloaded the last of the data from the last channel, we can change to the next state, `DIFFERENCE`:

```
// recover the address from the user channel
ram.waddr =
  7x{{2x{{mag_phase.m_axis_dout_tuser[addr.WIDTH-2:0]}}}};
ram.write_en[unload_ch_ctr.q] =
```

```
    2x{mag_phase.m_axis_dout_tvalid
    // write only first half of values
    & ~mag_phase.m_axis_dout_tuser[addr.WIDTH-1]};
  ram.write_data[unload_ch_ctr.q] =
    // phase, mag
    {mag_phase.m_axis_dout_tdata[31:16], mag_phase.m_axis_dout_tdata[15:0]};

  // if we have processed all the samples we need to
  if (mag_phase.m_axis_dout_tvalid && mag_phase.m_axis_dout_tlast) {
    unload_ch_ctr.d = unload_ch_ctr.q + 1;   // move onto the next channel
    // if we have processed all 7 channels
    if (unload_ch_ctr.q == 6) {
      state.d = state.DIFFERENCE;            // move to the next stage
      addr_pipe.d = 2x{{addr.WIDTHx{1b0}}};
    }
  }
```

DIFFERENCE State

In this stage, we subtract the phase of the center microphone from each of the six outer microphones. We then use this difference to scale the microphone's location vectors and then sum the vectors. The single resulting vector is fed through the CORDIC to get its angle, which is then written back to RAM.

The first operation that takes place is the subtraction of the phases. The only interesting part here is that we need to keep the resulting differences in the +/– pi range. This is done by checking for overflow and adding or subtracting 2pi. All the operations here are intended to be signed, so everything is wrapped in `$signed` again to ensure this:

```
  state.DIFFERENCE:
    for (i = 0; i < 6; i++) {
      // we care about the difference in phase between the center microphone
      // and the outer microphones
      // as this is proportional to the delay of the sound (divided by the
      // frequency)
      temp_phase[i] =
        $signed(ram.read_data[i][1]) - $signed(ram.read_data[6][1]);

      // we need to keep the difference in the +/- pi range for the next
      // steps

      // 25736 = pi (4.13 fixed point)
      if ($signed(temp_phase[i]) > $signed(16d25736)) {
        // 51472 = 2*pi (4.13 fixed point)
          temp_phase[i] = $signed(temp_phase[i]) - $signed(17d51472);
        } else if ($signed(temp_phase[i]) < $signed(-16d25736)) {
          temp_phase[i] = $signed(temp_phase[i]) + $signed(17d51472);
        }
      }
```

Because of all the multiplications, the scaling of the vectors and summing them happens over two clock cycles. This adds a little bit of complexity: because the CORDIC can tell us it can't accept new data at any time, we need to be able to stall the pipeline.

To do this, we detect the specific case when pipeline data would be lost and revert to the dropped address. The only time this can happen is if the pipeline has been active for at least two cycles before being halted. This method can cause some values to be calculated more than once, depending on the halt/resume patterns, but this isn't a big deal as long as every value is calculated at least once and we continue to make progress. The worst-case pattern would be run, run, halt, run, run, halt, and so on. In this case, we would advance only one address per run, run, halt cycle. However, we would still make progress and cover all the values. The actual behavior of the CORDIC will have many more halts between a single run, and with this pattern we don't repeat any values.

Some of the scaling is really simple; for example, the first microphone is at (–1, 0), so the x value is just the negated phase difference, and the y component is always 0. However, some require multiplying by sqrt(3)/2. This is done using fixed-point multiplication and a bit shift.

With all the scaled phase values calculated, they are summed and then divided by 8, which is the closest power of 2 greater than 6 (the number of microphones). The division is used to keep the values in a 16-bit range:

```
/* Sample coordinates

   0: (-1,    0)
   1: (-1/2,  sqrt(3)/2)
   2: ( 1/2,  sqrt(3)/2)
   3: ( 1     0)
   4: ( 1/2, -sqrt(3)/2)
   5: (-1/2, -sqrt(3)/2)
          6: ( 0,    0)
*/

addr_pipe.d[0] = addr.q; // output address of the ram

if (mag_phase.s_axis_cartesian_tready) {
  /*
     Here we are scaling each microphone's location vector by
     he delay (phase difference). This will give us a vector
     roportional to that microphone's contribution to the total
     direction.
  */
  scaled_phase.d[0][0] = -temp_phase[0];
  scaled_phase.d[0][1] = 0;

  mult_temp = -temp_phase[1];
  scaled_phase.d[1][0] = c{mult_temp[16],mult_temp[16:1]};
```

```
mult_temp = $signed(temp_phase[1]) * $signed(17d56756);
// phase * sqrt(3)/2
scaled_phase.d[1][1] = mult_temp[mult_temp.WIDTH-2:16];

scaled_phase.d[2][0] = c{temp_phase[2][16], temp_phase[2][16:1]};

mult_temp = $signed(temp_phase[2]) * $signed(17d56756);
scaled_phase.d[2][1] = mult_temp[mult_temp.WIDTH-2:16];

scaled_phase.d[3][0] = temp_phase[3];
scaled_phase.d[3][1] = 0;

scaled_phase.d[4][0] = c{temp_phase[4][16], temp_phase[4][16:1]};

mult_temp = $signed(temp_phase[4]) * $signed(-17d56756);
scaled_phase.d[4][1] = mult_temp[mult_temp.WIDTH-2:16];

mult_temp = -temp_phase[5];
scaled_phase.d[5][0] = c{mult_temp[16], mult_temp[16:1]};

mult_temp = $signed(temp_phase[5]) * $signed(-17d56756);
scaled_phase.d[5][1] = mult_temp[mult_temp.WIDTH-2:16];

addr_pipe.d[1] = addr_pipe.q[0]; // address of scaled vector values

/*
   With all the scaled vectors, we simply need to sum them to get
   the overall direction of sound for this frequency.
*/
summed_phase[0] = 0;
summed_phase[1] = 0;
for (i = 0; i < 6; i++) {
  summed_phase[0] = $signed(scaled_phase.q[i][0]) +
    $signed(summed_phase[0]);
  summed_phase[1] = $signed(scaled_phase.q[i][1]) +
    $signed(summed_phase[1]);
}

// if there are more samples to go, advance the addr
if (addr.q != SAMPLES/2)
  addr.d = addr.q + 1;

// use the summed vectors (divided by 8) to calculate the overall
// direction of sound
mag_phase.s_axis_cartesian_tdata =
  c{summed_phase[1][3+:16], summed_phase[0][3+:16]};
// only valid for the first half of addr
mag_phase.s_axis_cartesian_tvalid = ~addr_pipe.q[1][addr.WIDTH-1];

// feed in the address for later use
mag_phase.s_axis_cartesian_tuser = addr_pipe.q[1];
```

```
    mag_phase.s_axis_cartesian_tlast = addr_pipe.q[1] == SAMPLES/2 - 1;

  } else if (&mag_phase_ready.q) {
    // if we were ready but now aren't we need to go back an address so
    // that we don't skip one
    addr.d = addr_pipe.q[0];
  }
```

We now just need to feed the output of the CORDIC back into the RAM. This is basically the same as the last part of the `FFT` state:

```
  // write the phase data into the RAM channel 0
  ram.waddr = 7x{{2x{{mag_phase.m_axis_dout_tuser[addr.WIDTH-2:0]}}}};
  ram.write_data[0] =
    {mag_phase.m_axis_dout_tdata[31:16], mag_phase.m_axis_dout_tdata[15:0]};
  ram.write_en[0] = 2x{mag_phase.m_axis_dout_tvalid};

  // if we are on the last sample move onto the next stage
  if (mag_phase.m_axis_dout_tlast && mag_phase.m_axis_dout_tvalid) {
    addr.d = CROP_MIN;
    state.d = state.AGGREGATE_WAIT;
  }
```

AGGREGATE State

In this stage, we will run through the calculated directions and sum their corresponding magnitudes into 16 directional bins.

Even though we have 256 frequencies to work with, we will be summing only the ones between 9 and 199. The lowest frequencies aren't too useful, and the highest ones get out of hearing range. These values are set by `CROP_MIN` and `CROP_MAX`.

The bin selection is performed with a series of `if` statements that check whether the angle lies in a particular bin's range. Only the 8 MSBs are used to save on the size of the comparisons. It doesn't make a difference if the bin boundaries aren't perfectly precise. These comparisons are all signed, so the constants are wrapped in `$signed`. The signal `angle` is declared as `signed`, so the `$signed` function isn't required:

```
  state.AGGREGATE:
    addr.d = addr.q + 1;
    angle = ram.read_data[0][1][15:8]; // angle calculated in the last step
    magnitude = ram.read_data[6][0];   // use the magnitude from the center mic

    /*
       We now need to go through each frequency and bin them into one of 16 groups.
       This makes it easier to get an idea of where the sound is coming from as
       many frequencies will point in the same direction of a single source. If
       we have multiple sources then multiple bins will receive a lot of values.
       A more advanced grouping method could be done in software off chip such as
       K-means to get a more accurate picture, but this method works relatively
       well and is simple to implement in hardware.
```

```
*/
if (angle >= $signed(ANGLE_BOUNDS[7]) || angle < $signed(-ANGLE_BOUNDS[7])) {
  sums.d[0] = sums.q[0] + magnitude;
} else if (angle >= $signed(ANGLE_BOUNDS[6]) &&
           angle < $signed(ANGLE_BOUNDS[7])) {
  sums.d[1] = sums.q[1] + magnitude;
} else if (angle >= $signed(ANGLE_BOUNDS[5]) &&
           angle < $signed(ANGLE_BOUNDS[6])) {
  sums.d[2] = sums.q[2] + magnitude;
} else if (angle >= $signed(ANGLE_BOUNDS[4]) &&
           angle < $signed(ANGLE_BOUNDS[5])) {
  sums.d[3] = sums.q[3] + magnitude;
} else if (angle >= $signed(ANGLE_BOUNDS[3]) &&
           angle < $signed(ANGLE_BOUNDS[4])) {
  sums.d[4] = sums.q[4] + magnitude;
} else if (angle >= $signed(ANGLE_BOUNDS[2]) &&
           angle < $signed(ANGLE_BOUNDS[3])) {
  sums.d[5] = sums.q[5] + magnitude;
} else if (angle >= $signed(ANGLE_BOUNDS[1]) &&
           angle < $signed(ANGLE_BOUNDS[2])) {
  sums.d[6] = sums.q[6] + magnitude;
} else if (angle >= $signed(ANGLE_BOUNDS[0]) &&
           angle < $signed(ANGLE_BOUNDS[1])) {
  sums.d[7] = sums.q[7] + magnitude;
} else if (angle >= $signed(-ANGLE_BOUNDS[0]) &&
           angle < $signed(ANGLE_BOUNDS[0])) {
  sums.d[8] = sums.q[8] + magnitude;
} else if (angle >= $signed(-ANGLE_BOUNDS[1]) &&
           angle < $signed(-ANGLE_BOUNDS[0])) {
  sums.d[9] = sums.q[9] + magnitude;
} else if (angle >= $signed(-ANGLE_BOUNDS[2]) &&
           angle < $signed(-ANGLE_BOUNDS[1])) {
  sums.d[10] = sums.q[10] + magnitude;
} else if (angle >= $signed(-ANGLE_BOUNDS[3]) &&
           angle < $signed(-ANGLE_BOUNDS[2])) {
  sums.d[11] = sums.q[11] + magnitude;
} else if (angle >= $signed(-ANGLE_BOUNDS[4]) &&
           angle < $signed(-ANGLE_BOUNDS[3])) {
  sums.d[12] = sums.q[12] + magnitude;
} else if (angle >= $signed(-ANGLE_BOUNDS[5]) &&
           angle < $signed(-ANGLE_BOUNDS[4])) {
  sums.d[13] = sums.q[13] + magnitude;
} else if (angle >= $signed(-ANGLE_BOUNDS[6]) &&
           angle < $signed(-ANGLE_BOUNDS[5])) {
  sums.d[14] = sums.q[14] + magnitude;
} else {
  sums.d[15] = sums.q[15] + magnitude;
}

// stop once we reach the highest frequency to count (we only care about
// audible ones)
```

```
    if (addr.q == CROP_MAX)
      state.d = state.OUTPUT;
```

Finally, with the different bins full, we can output the values. We first check for overflow and saturate the bin if it did.

Upon completion, we return to idle and wait for the command to start the process all over again:

```
  state.OUTPUT:
    for (i = 0; i < 16; i++) {
      sum = sums.q[i][sums.q.WIDTH[1]-1:0];
      if (sum > 65535) // if it overflowed, saturate it
        sum = 65535;

      result[i] = sum[15:0]; // use the 16 LSBs for decent sensitivity
    }
    result_valid = 1;

    state.d = state.IDLE;
```

This project is a fairly complicated example, but hopefully it gives you an idea of some of the interesting things you can do with an FPGA.

CHAPTER 13
Lucid Reference

This chapter contains information on Lucid and can be used as a reference. Although a lot of Lucid's syntax and features are covered throughout the book, this chapter explains them in a more comprehensive way. This chapter is roughly organized from an outside-in approach, meaning we start from a full module declaration and work our way down to individual expressions and constants.

The templates are written using syntax similar to ANTLR grammar. Things in single quotes are string literals, meaning they must appear as is. A question mark after something means that it is optional. An asterisk following something means zero or more of that thing may be present. A plus following something means one or more of that thing may be present. Parentheses are used to group multiple tokens.

The following page is intended as a quick reference guide. It covers the most commonly used features of Lucid but is not an exhaustive reference. Refer to the rest of the chapter for more details on any specific feature. You may find it helpful to print the reference guide.

Lucid Quick Reference

Modules

```
module name #(
    PARAM = DEFAULT_VALUE : CONSTRAINT
  )(
    port_type name
  ) {
  SIGNAL_AND_MODULE_INSTANCES
  always {
    ALWAYS_STATEMENTS
  }
}
```

Assignment Blocks

```
.port_name(EXPR), #PARAM_NAME(CONST_EXPR) {
  SIGNAL_AND_MODULE_INSTANCES
}
```

Types

`input`	Port input, read-only
`output`	Port output, write-only
`inout`	Bidirectional port with subsignals: *read*, *write*, *enable*
`sig`	Acts as a way to connect expressions
`var`	Used in loops
`dff`	Stores data, subsignals *clk*, *rst*, *d*, *q*
`fsm`	Similar to a *dff* but with built-in constants
`const`	Constant value

Instantiation

DFF: `dff name (.clk(clk_sig));`

FSM: `fsm name = { STATE, STATE, ... };`

Module: `module name (#PARAM(CONST_EXPR), .port(EXPR));`

Always Statements

Assignment: `signal = EXPR;`

If: `if (EXPR) { ... } else { ... }`

Case:
```
case (EXPR) {
  CONST: ...
  default: ...
}
```

For: `for (INIT; CONDITION; STEP) { ... }`

Bit Selectors

Single index:	`[EXPR]`
Constant range:	`[MAX_CONST:MIN_CONST]`
Fixed-width increment:	`[START+:WIDTH_CONST]`
Fixed-width decrement:	`[START-:WIDTH_CONST]`

Functions

`$clog2(a)`	Performs ceiling of $\log_2$(a)
`$pow(a,b)`	Raises *a* to the *b* power
`$reverse(a)`	Reverses the indices of the outermost dimension of *a*
`$flatten(a)`	Collapses *a* into a 1D array
`$signed(a)`	Forces *a* to be interpreted as signed
`$unsigned(a)`	Forces *a* to be interpreted as unsigned

Numbers

Integers:	`0-9`
Decimal:	`Wd0-9`
Hexadecimal:	`Wh0-9 A-F x z`
Binary:	`Wb0 1 x z`
Strings:	`"TEXT"`

Arrays

Array builder:	`{ EXPR, EXPR, ... }`
Concatenation:	`c{ EXPR, EXPR, ... }`
Duplication:	`NUM x{ EXPR }`

Bitwise Operators

NOT:	`~EXPR`
AND:	`EXPR & EXPR`
OR:	`EXPR | EXPR`
XOR:	`EXPR ^ EXPR`
XNOR:	`EXPR ~^ EXPR`

Logical Operators

NOT:	`!EXPR`
AND:	`EXPR && EXPR`
OR:	`EXPR || EXPR`
Ternary:	`EXPR ? EXPR : EXPR`

Comparison Operators

Less than:	`EXPR < EXPR`
Greater than:	`EXPR > EXPR`
Less than or equal:	`EXPR <= EXPR`
Greater than or equal:	`EXPR >= EXPR`
Equal:	`EXPR == EXPR`
Not equal:	`EXPR != EXPR`

Reduction Operators

OR:	`|EXPR`
NOR:	`~|EXPR`
AND:	`&EXPR`
NAND:	`~&EXPR`
XOR:	`^EXPR`
XNOR:	`~^EXPR`

Math Operators

Negate:	`-EXPR`
Add:	`EXPR + EXPR`
Subtract:	`EXPR - EXPR`
Multiply:	`EXPR * EXPR`
Divide:	`EXPR / EXPR`
Shift right:	`EXPR >> NUM_BITS`
Signed shift right:	`EXPR >>> NUM_BITS`
Shift left:	`EXPR << NUM_BITS`
Signed shift left:	`EXPR <<< NUM_BITS`

Modules

Module declarations take place at the root of a file. Each file can have at most one module declaration, and it is convention for the module's name to match the file's name.

A module declaration takes the following form:

```
'module' name param_list? port_list module_body
```

The declaration starts with the `module` keyword followed by the name of this module. The `param_list` is optional and is followed by the `port_list` and the `module_body`.

A module's name must start with a lowercase letter. The rest of the name can contain lower- or uppercase letters, numbers, and underscores.

Here is an example of a bare-bones, but complete, module:

```
module my_module (
    input clk,  // clock
    input rst,  // reset
    output out
  ) {

  always {
    out = 0;
  }
}
```

Parameter Lists

The purpose of a *parameter list* is to specify parameters that are used to make your module more flexible. The values of the parameters can be changed for each instance of the module, but they are always constant values.

A `param_list` takes the following form:

```
'#(' param_dec (',' param_dec)* ')'
```

That is, a series of one or more `param_dec` separated by commas and enclosed in parentheses prefixed by a hash symbol.

Each `param_dec` takes the following form:

```
name ('=' expr)? (':' param_constraint)?
```

Here is an example of a `param_dec`:

```
SIZE = 8 : SIZE > 0
```

It starts with the parameter's name, which must start with a capital letter followed by only capital letters, numbers, and underscores. This is followed by an optional default value, which is any constant expression, and an optional `param_constraint` that is prefixed with a colon.

The `param_constraint` is a constant expression that can use the parameter being declared and any parameters declared before this one. This expression is evaluated with the parameter values set when the module is instantiated. If they fail (have a value of 0), an error will be thrown.

Because the default value and constraint are optional, the most minimal version is simply a name:

```
SIZE
```

Here is an example of a full `param_list` from the Counter component:

```
#(
  SIZE = 8 : SIZE > 0, // Width of the output
  DIV = 0  : DIV >= 0, // number of bits to use as divisor
  TOP = 0  : TOP >= 0, // max value, 0 = none

  // direction to count, use 1 for up and 0 for down
  UP = 1 : UP == 1 || UP == 0
)
```

Port Lists

Every module has a *port list*. This is where you list the inputs, outputs, and inouts of the module. It defines all the external connections.

The `port_list` takes the following form:

```
'(' port_dec (',' port_dec)* ')'
```

It is a list of `port_dec` separated by commas and enclosed in parentheses.

The `port_dec` takes the following form:

```
'signed'? ('input' | 'output' | 'inout') struct_type? name array_size?
```

Here is an example of a few `port_dec` statements:

```
output<Memory.master> memOut
inout sda
signed output value [8]
```

The `port_dec` starts with an optional signed type modifier. This will make the value be interpreted as two's complement. This is followed by the type: `input`, `output`, or `inout`. Each port can then be a struct specified by an optional `struct_type` and an array of potentially multiple dimensions specified by the optional array size. The

name of a port is the port's name and must start with a lowercase letter followed by upper- or lowercase letters, numbers, or underscores.

Each `port_dec` can be for an `input`, `output`, or `inout`. Inputs and outputs are fairly straightforward. Inputs can be read and get their value only from an external connection. Outputs can only be written. The `inout` type is for bidirectional signals and can be used only by the top-level module of your design or must be routed directly to an `inout` in the top-level module. The `inout` can be read and written to.

Here is an example of a full `port_list` from the Memory Arbiter component:

```
(
  input clk,                                    // clock
  input rst,                                    // reset
  input<Memory.slave> memIn,                    // memory inputs
  output<Memory.master> memOut,                 // memory outputs
  input<Memory.master> devIn [DEVICES],         // devices inputs
  output<Memory.slave> devOut [DEVICES]         // devices outputs
)
```

Note that you can use a module's parameters in the array sizes of the ports. In the preceding example, the parameter `DEVICES` is used to size `devIn` and `devOut`.

Module Body

A *module's body* consists of a list of statements enclosed in curly braces:

```
'{' statement* '}'
```

A *statement* can be any one of the following.

Constant declaration

Constant declarations are used to define a constant that can be used later. Wherever the constant is used, it will be replaced by the value it represents. The declaration takes the following form:

```
'const' name '=' expr ';'
```

Here's an example:

```
const MY_CONST = 12;
```

The *`name`* of a constant must start with a capital letter and can be followed by capital letters, numbers, and underscores. It can't contain lowercase letters.

Here, *`expr`* is an expression. The expression must have a constant value—that is, one that can be determined at synthesis time.

Variable declaration

Variable declarations are used to define one or more variables that can be used later. Variables are used to store temporary integer values that you won't see in your actual design. The most common use for them is as the index in a `for` statement.

The declaration takes the following form:

```
'var' name array_size? (',' name array_size?)* ';'
```

Here's an example:

```
var i, j, k;
```

Each variable declaration can define one or more variables separated by commas. Each variable requires a *name* that starts with a lowercase letter followed by lower- or uppercase letters, numbers, and underscores. The name is then followed by an optional array size.

Unlike other signals, a `var` can hold integers without being an array. An array of `vars` would act like an array of `ints` in a programming language.

Signal declaration

Signal declarations are used to define one or more signals. Signals should be thought of as wires in your design; they are used to carry a value from one expression to another. Inside an `always` block, they can be read and written. Their value will be whatever was last written in the `always` block, and they must be written before being read.

The declaration takes the following form:

```
'signed'? 'sig' struct_type? name array_size? (',' name array_size?)*
```

Here's an example:

```
sig mySig [16];
```

The type `sig` behaves similarly to other types. Without the optional `struct_type` or `array_size`, it will be a single bit wide and can hold a value of 0 or 1. If `struct_type` is present, it is used for all `sigs` declared in this line.

DFF declaration

DFF declarations are used to define one or more DFFs. A DFF acts as a basic unit of memory and is commonly used to split up combinational logic.

The declaration takes the following form:

```
'signed'? 'dff' struct_type? name array_size? inst_cons?
(',' name array_size? inst_cons?)*
```

Here are some examples:

```
dff ctr [32] (#INIT(32d15), .clk(clk));
signed dff value [8] (.clk(clk), .rst(rst));
dff<color> rgbValue;
```

Like `input`, `output`, and `inout` types, the `dff` type has an optional `struct_type` and `array_size` that will specify the size of the `dff`. However, unlike other simpler types, the `dff` acts more like a module instance. The `dff` has four ports: `clk`, `rst`, `d`, and `q`. The `clk` port is used as the clock and must be connected when the `dff` is declared. The `rst` port can optionally be connected. If it isn't connected, the DFF won't have a reset signal. Typically, if you don't need a reset, don't use it. The ports `d` and `q` are used to input a new value and read the current value, respectively. These are typically used inside an `always` block.

The `dff` type also has a few optional parameters. The `INIT` parameter can be used to set the initial value of the DFF when the FPGA starts up or the reset signal is asserted. The `IOB` parameter accepts a value of 0 or 1, and when 1, the DFF will be marked to be packed into an I/O buffer in the FPGA. This is useful when the DFF output goes directly out of the FPGA and you want it have minimal delay. If the FPGA being targeted doesn't support this, it will be ignored.

The optional `inst_cons` takes the following form:

```
'(' (param_con | port_con) (',' (param_con | port_con))* ')'
```

Here's an example:

```
(#INIT(32d15), .clk(clk), .rst(rst))
```

This is a list of parameter and port connections enclosed by parentheses and separated by commas. The details of the connections are in the following section.

Instance connections. The connections made to the ports of a DFF declaration, FSM declaration, or module instance are made using the following form:

```
'.' name '(' expr ')'
```

Here are some examples:

```
.dataIn(myData)
.value(8d12)
```

Here *`name`* is the name of the port to be connected to *`expr`*, and *`expr`* is an expression. The name must start with a lowercase letter and can be followed by lower- or uppercase letters, numbers, and underscores.

Parameter assignments follow a similar form but replace the period with a hash:

```
'#' name '(' expr ')'
```

Here's an example:

```
#WIDTH(16)
```

Another difference is that *expr* must be known at synthesis time. That means it can contain only constants or parameters.

FSM declaration

FSM declarations are used to define a single FSM. The `fsm` type is similar to the `dff` type. The only real difference is that you specify a list of different constant states that can be assigned to it. The size of the `fsm` will be automatically set to be able to contain any of the states you declared. An array of `fsms` is able to store multiple states.

The declaration takes the following form:

```
'fsm' name array_size? inst_cons? '=' '{' name (',' name)* '}'
```

Here's an example:

```
fsm state (.clk(clk), .rst(rst)) =
  {IDLE, START, WAIT_CMD, READ, WRITE, STOP, WAIT} ;
```

The first *name* is the name of the `fsm`. This must start with a lowercase letter and can be followed by lower- or uppercase letters, numbers, and underscores.

The list of *names* is the list of states. These must start with an uppercase letter and can be followed by uppercase letters, numbers, and underscores.

Like DFF declarations, `fsms` have the `INIT` parameter. However, if present, it can be assigned to only one of the states. If it isn't present, the first state in the list is used by default as the initial state.

Again like `dffs`, there are four ports: `clk`, `rst`, `d`, and `q`, which behave exactly the same way.

To access the state constants of an `fsm`, you prefix the name with the name of the `fsm` followed by a period. For example, if we had an `fsm` called `current_state` and it had a state named `RUNNING`, we could access this constant with `current_state.RUNNING`. This allows you to use the same state names for different `fsms` without collision.

Struct declaration

Struct declarations allow you to define a new `struct`. The `struct` can then be used to size other signals later in your design. A `struct` allows you to bundle a collection of names into a single type. It is similar to an array, but the elements are accessed with names instead of numbers and each element can have a unique width.

The declaration takes the following form:

```
'struct' name
  '{' name struct_type? array_size? (',' name struct_type? array_size?)* '}'
```

Here's an example from the Memory Arbiter component:

```
// simple structure to hold pending commands
struct command {
  valid,                          // valid flag (1 = valid)
  write,                          // write/read flag (1 = write)
  addr[23],                       // address to read/write
  data[32]                        // data for writes
}
```

The *name* of the `struct` must start with a lowercase letter and can be followed by lower- or uppercase letters, numbers, and underscores.

Following the *name* is a list of the `struct`'s members. Each member can simply be a name, which follows the same naming convention as the `struct`. However, the members can also be arrays and/or `structs` allowing you to nest `structs`.

Here is an example of using nested `structs`:

```
struct color {
  red[8],
  green[8],
  blue[8]
}

struct video_data {
  valid,
  pixel<color>
}

sig mySig<video_data>;

always {
  mySig.pixel.red = 129;
}
```

If two data types are declared with the same `struct` type, they can be directly assigned to each other. This is equivalent to assigning each member individually.

Module instance

Module instantiation is similar to using DFF declaration, except a module in your project serves as the *type*. An instantiation takes the following form:

```
name name array_size? inst_cons?
```

Here are some examples:

```
reset_conditioner reset_cond;
pipeline pipe [8] (#DEPTH(16));
```

The first *name* is the name of the module you want to instantiate, and the second *name* is the name of this instance. This must start with a lowercase letter and can be followed by lower- or uppercase letters, numbers, and underscores.

Similar to using DFF declarations, you can connect inputs and parameters to the module with the optional instance connections.

Inputs that weren't assigned at instantiation and outputs can be accessed using the name of the instance followed by a period and the name of the port—for example, `counter.value`.

You can also specify an array size for the instance. If you do this, that module will be duplicated in your design for each element in the array. Any connections or parameters specified at instantiation will be connected to each instance separately. For example, if your module has a single-bit input `clk` and you connect a signal to it, that single-bit signal will be duplicated and connected to each module. However, ports not connected at instantiation are packed into arrays. Single-bit signals become arrays the same size as your module array, and arrays become multidimensional arrays. The module indices become the first dimensions in the array. In other words, you select the module first and then the bits in the port.

Assignment block

An *assignment block* is a way to connect a signal or value to a port or parameters of multiple modules. It is most commonly used to connect the clock and reset signals as they exist on most modules.

The assignment block takes the following form:

```
con_list '{' ((dff_dec | fsm_dec | module_inst) ';' | assign_block)* '}'
```

Here `con_list` is a comma-separated list of instance connections. These are the parameter and port connections that will be applied to all the DFF declarations, FSM declarations, and module instances. All instances in the block must have ports and parameters with the same names as the ones in the list.

Assignment blocks can be nested.

Here is an example of a common use case:

```
.clk(clk) {
  dff noResetDff;
  dff anotherDff;

  .rst(rst) {
```

```
    dff dffWithReset;

    myModule instOfMyModule;
  }
}
```

always block

`always` blocks are where all a module's logic resides. This is where you can assign values to signals, perform calculations, and evaluate expressions.

An `always` block takes the following form:

```
'always' block
```

It simply is the `always` keyword followed by a *`block`*. A *`block`* is a single `always` statement, or multiple `always` statements enclosed in curly brackets.

Statements are evaluated from top to bottom, with lower statements having precedence over previous statements. In this regard, it can feel a bit like programming, where statements are evaluated sequentially. However, they aren't really evaluated sequentially, but simply interpreted sequentially. You should think of everything in an `always` block as being evaluated continuously.

The statements in an `always` block can be any of the following.

Assignment. Assignments are the most basic `always` statement and are perhaps the most useful. They take the following form:

```
signal '=' expr ';'
```

Here are some examples:

```
data_out = data.q;
scl_reg.d = 0;
scl_reg.d = c{2b11, (scl_reg.WIDTH-2)x{1b0}} + 1;
state.d = state.WRITE;
```

Here *`signal`* is anything that can be assigned a value, and *`expr`* is an expression.

As you might expect, this will assign the value of *`expr`* to *`signal`*.

If statement. `If` statements provide a simple method for making decisions. They allow you to make a set of `always` statements to be conditionally evaluated.

An `if` statement takes the following form:

```
'if' '(' expr ')' block ('else' block)?
```

Here's an example:

```
if (!&ctr.q)         // if counter isn't full
  ctr.d = ctr.q + 1; // increment it
```

```
else
  ctr.d = newValue;
```

If the value of *expr*, which is an expression, is nonzero, then the first *block* is evaluated. If the value of *expr* is zero and the optional `else` clause is present, then the second *block* is evaluated.

A *block* is a single `always` statement, or multiple `always` statements enclosed in curly brackets.

Case statement. Case statements provide a compact way to select a set of `always` statements based on the value of an expression.

They take the following form:

```
'case' '(' expr ')' '{' case_elem+ '}'
```

Here's an example:

```
case (ctr.q) {
  0: out = 15;
  1: out = 6;
  2: out = 24;
  default: out = 0;
}
```

Here *expr* is the expression to evaluate and is an expression.

Each `case_elem` takes the following form:

```
(expr | 'default') ':' always_statements+
```

Here's an example:

```
state.INIT:
  value = 1;
  ctr.d = 0;
  state.d = state.NEXT;
```

This *expr* is also an expression, but it must have a constant value known at synthesis time. The `default` keyword is used to catch all values not otherwise stated.

If the *expr* being evaluated by the `case` statement matches the `case_elem's` *expr* value, then the corresponding set of `always` statements are evaluated.

The same effect could be accomplished using a set of `if-else` statements, but `case` statements are more concise. Unlike programming, there is no performance improvement generally associated with using a case statement over a series of `if-else` statements.

for statement. `for` statements provide a compact way to write something that is otherwise repetitive. It takes the following form:

```
'for' '(' assign_stat ';' expr ';' var_assign ')' block
```

Here's an example from the Encoder component:

```
for (i = 0; i < WIDTH; i++) {
  if (in[i]) // if the bit is set
    out = i; // the output is the index of that bit
}
```

A `for` loop has four main components. The *`assign_stat`* is an assignment that is evaluated once before everything else. The *`expr`*, an expression, is evaluated before each loop iteration. If it is nonzero, the loop continues. Otherwise, the loop exits. The *`var_assign`* is evaluated at the end of each iteration. The *`var_assign`* is identical to an assignment, except that it also supports the *increment* and *decrement* shorthand for variables.

The form for variable increment and decrement is as follows:

```
signal ('++'|'--')
```

Here's an example:

```
i++
```

Here, *`signal`* is a variable. If it is followed by `++`, the value of the variable is incremented by 1. If it is followed by `--`, it is decremented by 1.

Each iteration of the loop, the statements in *`block`* are evaluated with a different value of the loop variable. The loop will behave exactly the same as if you manually copied and pasted the block and replaced any instances of the loop variable with each loop's value. `For` loops must have a constant number of iterations, because if the loop can't be unrolled, it can't be realized in hardware.

Expressions

Expressions are common throughout Lucid. Although there are many operators, they all boil down to a single value. The following is a list of the possible forms for an expression.

Signals and Constants

The most basic form of an expression is a *signal* or *constant*. This can be the name of any readable signal such as inputs, module instance outputs, sigs, vars, parameters, or constants. It can also be a literal value.

Functions

Functions perform an operation on a set of arguments. Many of them can be used only with constant values and are evaluated at synthesis. Others simply change how values are interpreted.

$clog2(arg)

This function performs the operation *ceiling(log_2(arg))*—that is, the ceiling of the log base 2 of its argument. This is useful when needing to determine the number of bits that are needed to store a certain number, as *`$clog2(num + 1)`* will return the minimum numbers of bits to store the value *`num`* with the exception of *`num`* being 0. This function can be used only with constant values and is evaluated at synthesis time.

$pow(arg1, arg2)

This function performs the operation $arg1^{arg2}$. That is, it raises *`arg1`* to the power of *`arg2`*. This function can be used only with constant values and is evaluated at synthesis time.

$reverse(arg)

This function takes the outermost index of the array *`arg`* and reverses it. The most common use is for reversing the indices of a string to make the leftmost letter index 0. This function can be used only with constant values and is evaluated at synthesis time.

$flatten(arg)

This function takes the array *`arg`* and reduces it to a single-dimensional array. For example, an 8 x 2 array would become a 16-bit array. It would be arranged with the 2 LSBs being the 2 bits indexed by 0 for the outermost dimension. This can be used on any signals.

$signed(arg)

This function causes *`arg`* to be interpreted as a signed (two's complement) value. This can be used on any signals.

$unsigned(arg)

This function causes *`arg`* to be interpreted as an unsigned value. This can be used on any signals.

Groups

Groups are indicated by parentheses; we group an expression so that it is evaluated before things outside the parentheses. It takes the following form:

```
'(' expr ')'
```

Array Concatenation

Array concatenation allows you to concatenate two arrays as long as all their dimensions match, excluding the outermost one. For example, you can concatenate a 3 x 8 array with a 4 x 8 array to make a 7 x 8 array. It takes the following form:

```
'c{' expr (',' expr)* '}'
```

Array Duplication

Array duplication allows you to duplicate the outermost dimension of an array. It has the same effect as concatenating an array with itself multiple times. It takes the following form:

```
expr 'x{' expr '}'
```

The first *expr* specifies the number of times the outermost dimension of the array, the second *expr*, should be duplicated. The first *expr* must be a constant expression that can be evaluated at synthesis time.

Array Builder

The *array builder* is used to pack multiple values into an array. Each value must have the same dimensions. It takes the following form:

```
'{' expr (',' expr)* '}'
```

The dimensions of the new array will be one higher than the dimension of any element.

Bitwise Invert

The *bitwise inverting operator* is used to flip the values of all the bits in an expression. In other words, all 0s become 1s, and all 1s become zeros. It takes the following form:

```
'~' expr
```

Logical Invert

The *logical inverting operator* is used to negate the logical value of an expression. If the expression is nonzero, it becomes 0. If it is 0, it becomes 1. It takes the following form:

```
'!' expr
```

Multidimensional arrays are considered to be true if any element is nonzero.

Negate

The *negation operator* performs a two's complement negation on the expression. It takes the following form:

```
'-' expr
```

Multiply

The *multiplication operator* multiplies two values and takes the following form:

```
expr '*' expr
```

Divide

The *division operator* divides two values and takes the following form:

```
expr '/' expr
```

While the division operator *can* be used on signals, it is highly recommended to use this operator only when the result is constant and can be evaluated at synthesis time. If you use it with a dynamic value, the resulting circuit is expensive. Instead, if you need to perform division, you should use Xilinx's CoreGen tool to generate a division module with optimal parameters for your design.

Add and Subtract

Addition and subtraction operators add or subtract two numbers, respectively. They take the following form:

```
expr ('+'|'-') expr
```

When the `+` symbol is used, the expressions are added together. When the `-` symbol is used, the second expression is subtracted from the first.

Bit Shifting

Bit shifting allows you to shift the bits in a single-dimensional array either left or right. It takes the following form:

```
expr ('>>'|'<<'|'>>>'|'<<<') expr
```

The first *expr* is shifted by the value of the second *expr*.

If the >> operator is used, the array is shifted right and padded with 0s.

If the << operator is used, the array is shifted left and padded with 0s.

If the >>> operator is used, the array is shifted right and padded with 0s if the value being shifted is unsigned. If the value is signed, it is padded with the value of the MSB.

If the <<< operator is used, the array is shifted left and padded with 0s.

Bitwise Operators

The *bitwise operators* perform a Boolean operation on two arrays bit by bit. The two arrays must have matching dimensions.

It takes the following form:

```
expr ('|'|'&'|'^'|'~^') expr
```

The | operator performs a bitwise OR.

The & operator performs a bitwise AND.

The ^ operator performs a bitwise XOR.

The ~^ operator performs a bitwise XNOR.

The resulting array has the same dimensions as both inputs.

Reduction Operators

Reduction operators take in only one argument and produce a single bit. They simply perform a Boolean operation on all the bits in an array.

It takes the following form:

```
('&'|'~&'|'|'|'~|'|'^'|'~^') expr
```

The | operator ORs all the bits together.

The ~| operator NORs all the bits together.

The & operator ANDs all the bits together.

The `~&` operator NANDs all the bits together.

The `^` operator XORs all the bits together.

The `~^` operator XNORs all the bits together.

Comparison Operators

The *comparison operators* allow you to compare two values and create a Boolean value (1 or 0).

It takes the following form:

```
expr ('<'|'>'|'=='|'!='|'<='|'>=') expr
```

The `<` operator is true when the first *expr* is less than the second *expr*.

The `>` operator is true when the first *expr* is greater than the second *expr*.

The `==` operator is true when the first *expr* is equal to the second *expr*.

The `==` operator is true when the first *expr* is not equal to the second *expr*.

The `<=` operator is true when the first *expr* is less than or equal to the second *expr*.

The `>=` operator is true when the first *expr* is greater than or equal to the second *expr*.

Logical AND and OR

The *logical operators* are used to combine multiple logical expressions. True is considered anything nonzero, and false is 0.

They take the following form:

```
expr ('||'|'&&') expr
```

The `||` operator will return 1 when either *expr* is nonzero, and 0 otherwise.

The `&&` operator will return 1 only when both *expr* are nonzero, and 0 otherwise.

Ternary Operator

The *ternary operator* is basically a compact `if` statement. It allows you to select one of two values based on the logical value of another.

It takes the following form:

```
expr '?' expr ':' expr
```

If the first *expr* is nonzero, the expression takes the value of the second *expr*. However, if it is 0, the expression takes the value of the third *expr*.

The second and third *expr* must be the same size because the result of the expression must be a fixed size.

Literal Values

Literal values can be expressed using numbers in binary, decimal, or hexadecimal as well as strings.

Numbers

Numbers can be specified in binary, decimal, or hexadecimal. The width used to represent the number can be explicitly or implicitly stated.

The most basic way to represent a value is to type it, such as `12`. This defaults to decimal with an implicit width. When the width is implicit, the value will have the minimum number of bits needed to represent the value.

To specify a radix, you prefix the value with `b` for binary, `d` for decimal, or `h` for hexadecimal. For example, `hF2` or `b110110`.

To specify a width, you must specify the radix first and then prefix the whole thing with the number of bits to use. For example, `8h2C` or `12d8`.

When using binary or hexadecimal, you can also use the characters `x` for *don't care* and `z` for *high impedance*.

Strings

Strings are an easy way to get the ASCII values for characters. A *string* consists of double quotes enclosing a set of characters. A single character string, such as `"B"`, is an 8-bit array. However, strings with more than one character become two-dimensional arrays, with the first dimension being the letter index, and the second dimension is the bits of each letter. It is therefore an *N* x 8 array, where *N* is the number of letters in the string. This makes it easy to access the individual letters.

The first letter in the string is the rightmost one. This is opposite of how we read them, but consistent with the right being the least significant value. Because of this, it is common to use `$reverse(arg)` to make the leftmost letter index 0.

global Blocks

`global` blocks give you a way to declare constants and structs that can be used anywhere in your design. They are especially helpful because they allow you to use structs in the ports of your modules. A file can contain any number of `global` blocks.

Each block is named, and that name is used when accessing the constants. The `global` block's name must be unique throughout your design.

They take the following form:

```
'global' name '{' (struct_dec | const_dec)* '}'
```

The *name* of a `global` block must begin with an uppercase letter and can be followed by lower- or uppercase letters, numbers, and underscores. It must also contain at least one lowercase letter.

See "Constant declaration" on page 187 and "Struct declaration" on page 190 for `const_dec` and `struct_dec`, respectively.

To access a globally declared struct or constant, start with the `global` block's name followed by a period and then the name of the constant or struct. See the following example from the OV2640 Interface component:

```
global Camera {
  // structure for storing the image data
  struct image_data {
    end_frame,        // end of frame reached (active high)
    end_line,         // end of line reached (active high)
    new_pixel,        // new pixel (active high)
    pixel [16]        // pixel data (valid when new_pixel = 1)
  }
}

...

    output<Camera.image_data> image // image data
```

Array Size and Bit Selection

Array sizes are used when declaring many types in Lucid to turn them into arrays. *Bit selection* is used to index the various elements inside an array.

Array Size

Array sizes are pretty simple and take the following form:

```
('[' expr ']')*
```

Here *expr* is an expression, but it must have a value that can be determined at synthesis time.

If no sizes are specified, the type is assumed to be 1 bit or one instance. For each size specified, the array will get another dimension. For example, something declared with `[4][8]` would be considered a 4 x 8 two-dimensional array.

Bit Selectors

Bit selectors are used to index into arrays. The most basic kind, simple indices, select a single element from the array. The more complex constant bit and fixed-width bit selectors allow selection of multiple elements.

When indexing into a multidimensional array, only the last index can be a constant bit selector or a fixed-width bit selectors. Every other index must be a simple array index.

Array index

The array index is the simplest form of indexing into an array. It selects a single element and takes the following form:

```
'[' expr ']'
```

Here *expr* is an expression. When indexing into a multidimensional array, the first index specified will index into the leftmost dimension when the array was declared. For example, `sig mySig[4][7]` can be indexed later with `mySig[3]`. This will have a width of 7. To index to a single bit, you could use `mySig[3][6]`.

Constant bit selector

Constant bit selectors allow you to select everything between a fixed start and stop indices. It takes the following form:

```
'[' expr ':' expr ']'
```

Both *expr* elements are expressions, but they both must be constant values that can be determined at synthesis time.

The first *expr* must be greater or equal to the second *expr* in value. These specify an inclusive range to select from the array.

Fixed-width bit selector

The fixed-width bit selector allows you to specify a start index and the number of elements to select above or below that index. The benefit of this over the constant index selector is that the start index doesn't need to be a constant value.

It takes the following form:

```
'[' expr ('+'|'-') ':' expr ']'
```

Again, both *expr* elements are expressions. The first *expr* is the starting index. The second *expr* is the number of elements to select. If the selector uses the `+` symbol, it selects that many elements above the starting index. If it is a `-` symbol, it selects that many elements below the starting index. The starting index itself is always included.

Comments

Comments give you a way to annotate your designs. The tools will ignore everything in comments. Lucid uses C/Java style comments, with single-line and block comments available.

Single-line comments start after `//` and end at the end of the line. Block comments start with `/*` and end with `*/`. These can span multiple lines or begin and end in the same line.

Full Modules and Proof

Greeter Module

This is the full `greeter` module greeting someone over the USB port. Referenced from "Getting Personal with RAM" on page 95:

```
module greeter (
    input clk,         // clock
    input rst,         // reset
    input new_rx,      // new RX flag
    input rx_data[8],  // RX data
    output new_tx,     // new TX flag
    output tx_data[8], // TX data
    input tx_busy      // TX is busy flag
  ) {

  // reverse so index 0 is the left most letter
  const HELLO_TEXT = $reverse("\r\nHello @!\r\n");
  const PROMPT_TEXT = $reverse("Please type your name: ");

  .clk(clk) {
    .rst(rst) {
      fsm state = {IDLE, PROMPT, LISTEN, HELLO}; // our state machine
    }

    // we need our counters to be large enough to store all the indices of our
    // text

    // HELLO_TEXT is 2D so WIDTH[0] gets the first dimension
    dff hello_count[$clog2(HELLO_TEXT.WIDTH[0])];
    dff prompt_count[$clog2(PROMPT_TEXT.WIDTH[0])];

    dff name_count[5]; // 5 allows for 2^5 = 32 letters
    // we need our RAM to have an entry for every value of name_count
```

```
  simple_ram ram (#SIZE(8), #DEPTH($pow(2,name_count.WIDTH)));
}


always {
  ram.address = name_count.q; // use name_count as the address
  ram.write_data = 8hxx;      // don't care
  ram.write_en = 0;           // read by default

  new_tx = 0;                 // default to no new data
  tx_data = 8hxx;             // don't care

  case (state.q) { // our FSM
    // IDLE: Reset everything and wait for a new byte.
    state.IDLE:
      hello_count.d = 0;
      prompt_count.d = 0;
      name_count.d = 0;
      if (new_rx) // wait for any letter
        state.d = state.PROMPT;

    // PROMPT: Print out name prompt.
    state.PROMPT:
      if (!tx_busy) {
        prompt_count.d = prompt_count.q + 1;   // move to the next letter
        new_tx = 1;                            // send data
        tx_data = PROMPT_TEXT[prompt_count.q]; // send letter from PROMPT_TEXT
        if (prompt_count.q == PROMPT_TEXT.WIDTH[0] - 1) // no more letters
          state.d = state.LISTEN;              // change states
      }

    // LISTEN: Listen to the user as they type his/her name.
    state.LISTEN:
      if (new_rx) { // wait for a new byte
        ram.write_data = rx_data;        // write the received letter to RAM
        ram.write_en = 1;                // signal we want to write
        name_count.d = name_count.q + 1; // increment the address

        // We aren't checking tx_busy here that means if someone types super
        // fast we could drop bytes. In practice this doesn't happen.

        // only echo non-new line characters
        new_tx = rx_data != "\n" && rx_data != "\r";
        tx_data = rx_data; // echo text back so you can see what you type

        // if we run out of space or they pressed enter
        if (&name_count.q || rx_data == "\n" || rx_data == "\r") {
          state.d = state.HELLO;
          name_count.d = 0; // reset name_count
        }
      }
```

```
      // HELLO: Prints the hello text with the given name inserted
      state.HELLO:
        if (!tx_busy) { // wait for tx to not be busy
          if (HELLO_TEXT[hello_count.q] != "@") { // if we are not at the sentry
            hello_count.d = hello_count.q + 1;    // increment to next letter
            new_tx = 1;                           // new data to send
            tx_data = HELLO_TEXT[hello_count.q];  // send the letter
          } else {                                // we are at the sentry
            // increment the name_count letter
            name_count.d = name_count.q + 1;

            // if we are not at the end
            if (ram.read_data != "\n" && ram.read_data != "\r")
              new_tx = 1;                                     // send data

            // send the letter from the RAM
            tx_data = ram.read_data;

            // if we are at the end of the name or out of letters to send
            if (ram.read_data == "\n" || ram.read_data == "\r" || &name_count.q)
            {
              // increment hello_count to pass the sentry
              hello_count.d = hello_count.q + 1;
            }
          }

          // if we have sent all of HELLO_TEXT
          if (hello_count.q == HELLO_TEXT.WIDTH[0] - 1)
            state.d = state.IDLE; // return to IDLE
        }
    }
  }
}
```

ADC Multichannel Example

This is the full `mojo_top` module for reading all the ADC channels and displaying them on the LEDs using PWM. Referenced from Chapter 8:

```
module mojo_top (
    input clk,              // 50MHz clock
    input rst_n,            // reset button (active low)
    output led [8],         // 8 user controllable LEDs
    input cclk,             // configuration clock, AVR ready when high
    output spi_miso,        // AVR SPI MISO
    input spi_ss,           // AVR SPI Slave Select
    input spi_mosi,         // AVR SPI MOSI
    input spi_sck,          // AVR SPI Clock
    output spi_channel [4], // AVR general purpose pins
    input avr_tx,           // AVR TX (FPGA RX)
    output avr_rx,          // AVR RX (FPGA TX)
    input avr_rx_busy       // AVR RX buffer full
```

```
) {

sig rst;                  // reset signal

.clk(clk) {
  // The reset conditioner is used to synchronize the reset signal to the FPGA
  // clock. This ensures the entire FPGA comes out of reset at the same time.
  reset_conditioner reset_cond;

  .rst(rst){
    // the avr_interface module is used to talk to the AVR for access
    // to the USB port and analog pins
    avr_interface avr;

    pwm pwm[8]; // PWM modules for the LEDs
    dff channel_ctr[3];
  }
}

sig real_channel[4];
sig sample_channel[3];

always {
  reset_cond.in = ~rst_n; // input raw inverted reset signal
  rst = reset_cond.out;   // conditioned reset

  real_channel = channel_ctr.q > 1 ? channel_ctr.q + 2 : channel_ctr.q;
  sample_channel = avr.sample_channel > 1 ?
    avr.sample_channel - 2 : avr.sample_channel;

  if (avr.new_sample && sample_channel == channel_ctr.q)
    channel_ctr.d = channel_ctr.q + 1;

  // connect inputs of avr
  avr.cclk = cclk;
  avr.spi_ss = spi_ss;
  avr.spi_mosi = spi_mosi;
  avr.spi_sck = spi_sck;
  avr.rx = avr_tx;
  avr.channel = real_channel; // ADC set to channel 0
  avr.tx_block = avr_rx_busy; // block TX when AVR is busy

  // connect outputs of avr
  spi_miso = avr.spi_miso;
  spi_channel = avr.spi_channel;
  avr_rx = avr.tx;

  // unused serial port
  avr.tx_data = 8hxx;  // don't care
  avr.new_tx_data = 0; // no data

  pwm.update = 8h00;       // default values
```

```
    pwm.value = 8x{{8hxx}};  // 8 by 8 array of x's

    // update when we have a new sample
    pwm.update[sample_channel] = avr.new_sample;
    pwm.value[sample_channel] = avr.sample[9:2]; // use the 8 MSBs

    led = pwm.pulse;         // connect PWM output to all the LEDs
  }
}
```

Basic Processor

This is the module containing the meat of the processor referenced from Chapter 9:

```
global Inst {
  const NOP   = 4d0;  // 0 filled
  const LOAD  = 4d1;  // dest, op1, offset  : R[dest] = M[R[op1] + offset]
  const STORE = 4d2;  // src, op1, offset   : M[R[op1] + offset] = R[src]
  c SET   = 4d3;  // dest, c        : R[dest] = c
  const LT    = 4d4;  // dest, op1, op2     : R[dest] = R[op1] < R[op2]
  const EQ    = 4d5;  // dest, op1, op2     : R[dest] = R[op1] == R[op2]
  c BEQ   = 4d6;  // op1, c         : R[0] = R[0] + (R[op1] == c ? 2 : 1)
  c BNEQ  = 4d7;  // op1, c         : R[0] = R[0] + (R[op1] != c ? 2 : 1)
  const ADD   = 4d8;  // dest, op1, op2     : R[dest] = R[op1] + R[op2]
  const SUB   = 4d9;  // dest, op1, op2     : R[dest] = R[op1] - R[op2]
  const SHL   = 4d10; // dest, op1, op2     : R[dest] = R[op1] << R[op2]
  const SHR   = 4d11; // dest, op1, op2     : R[dest] = R[op1] >> R[op2]
  const AND   = 4d12; // dest, op1, op2     : R[dest] = R[op1] & R[op2]
  const OR    = 4d13; // dest, op1, op2     : R[dest] = R[op1] | R[op2]
  const INV   = 4d14; // dest, op1          : R[dest] = ~R[op1]
  const XOR   = 4d15; // dest, op1, op2     : R[dest] = R[op1] ^ R[op2]
}

module cpu (
    input clk,         // clock
    input rst,         // reset
    output write,      // CPU write request
    output read,       // CPU read request
    output address[8], // read/write address
    output dout[8],    // write data
    input din[8]       // read data
  ) {

  .clk(clk), .rst(rst) {
    dff reg[16][8]; // CPU Registers
  }

  instRom instRom;  // program ROM

  sig op[4];        // opcode
  sig arg1[4];      // first arg
  sig arg2[4];      // second arg
```

```
sig dest[4];      // destination arg
sig constant[8];  // constant arg

always {
  // defaults
  write = 0;      // don't write
  read = 0;       // don't read
  address = 8hxx; // don't care
  dout = 8hxx;    // don't care

  instRom.address = reg.q[0];   // reg 0 is program counter
  reg.d[0] = reg.q[0] + 1;      // increment PC by default

  op = instRom.inst[15:12];     // opcode is top 4 bits
  dest = instRom.inst[11:8];    // dest is next 4 bits
  arg1 = instRom.inst[7:4];     // arg1 is next 4 bits
  arg2 = instRom.inst[3:0];     // arg2 is last 4 bits
  constant = instRom.inst[7:0]; // constant is last 8 bits

  // Perform the operation
  case (op) {
    Inst.LOAD:
      read = 1;                                 // request a read
      reg.d[dest] = din;                        // save the data
      address = reg.q[arg1] + arg2;             // set the address
    Inst.STORE:
      write = 1;                                // request a write
      dout = reg.q[dest];                       // output the data
      address = reg.q[arg1] + arg2;             // set the address
    Inst.SET:
      reg.d[dest] = constant;                   // set the reg to constant
    Inst.LT:
      reg.d[dest] = reg.q[arg1] < reg.q[arg2];  // less than comparison
    Inst.EQ:
      reg.d[dest] = reg.q[arg1] == reg.q[arg2]; // equals comparison
    Inst.BEQ:
      if (reg.q[dest] == constant)              // if R[dest] == constant
        reg.d[0] = reg.q[0] + 2;                // skip next instruction
    Inst.BNEQ:
      if (reg.q[dest] != constant)              // if R[dest] != constant
        reg.d[0] = reg.q[0] + 2;                // skip next instruction
    Inst.ADD:
      reg.d[dest] = reg.q[arg1] + reg.q[arg2];  // addition
    Inst.SUB:
      reg.d[dest] = reg.q[arg1] - reg.q[arg2];  // subtraction
    Inst.SHL:
      reg.d[dest] = reg.q[arg1] << reg.q[arg2]; // shift left
    Inst.SHR:
      reg.d[dest] = reg.q[arg1] >> reg.q[arg2]; // shift right
    Inst.AND:
      reg.d[dest] = reg.q[arg1] & reg.q[arg2];  // bitwise AND
    Inst.OR:
```

```
            reg.d[dest] = reg.q[arg1] | reg.q[arg2];   // bitwise OR
          Inst.INV:
            reg.d[dest] = ~reg.q[arg1];                // bitwise invert
          Inst.XOR:
            reg.d[dest] = reg.q[arg1] ^ reg.q[arg2];   // bitwise XOR
        }
      }
    }
```

Direction Detection Proof

The following is a proof for the sound-direction detector referenced in Chapter 12.

First we need to look at the delay each microphone will see when rotated at angle Φ. Because we are measuring the delay in respect to the center microphone, it is the y component of its location multiplied by a constant. The constant will be the placeholder for the speed of sound and radius of the microphone ring. It isn't really that important because it will cancel later:

$$d_1 = C \cdot \sin(\Phi)$$

$$d_2 = C \cdot \sin\left(\Phi + \frac{2\pi}{3}\right)$$

$$d_3 = C \cdot \sin\left(\Phi - \frac{2\pi}{3}\right)$$

The x components of each of the three microphones are as follows:

$$x_1 = \cos(\Phi)$$

$$x_2 = \cos\left(\Phi + \frac{2\pi}{3}\right)$$

$$x_3 = \cos\left(\Phi - \frac{2\pi}{3}\right)$$

We now need to show that if we scale the x components by their respective delays and sum them up, they sum to 0:

$$C \cdot \sin\left(\Phi\right) \cdot \cos\left(\Phi\right) + C \cdot \sin\left(\Phi + \frac{2\pi}{3}\right) \cdot \cos\left(\Phi + \frac{2\pi}{3}\right) + C \cdot \sin\left(\Phi - \frac{2\pi}{3}\right) \cdot \cos\left(\Phi - \frac{2\pi}{3}\right) = 0$$

To show this is true, we need two trigonometry identities:

$$\sin\left(u\right) \cdot \cos\left(v\right) = \frac{1}{2}[\sin(u + v) + \sin(u - v)]$$

$$\sin(u \pm v) = \sin(u) \cdot \cos(v) \pm \cos(u) \cdot \sin(v)$$

We can now divide out C and apply the first identity to the three sine-cosine pairs. Note that sin(0) is 0, so the second term of the identity is 0 for all three as u and v are the same:

$$\frac{1}{2}\sin\left(2\Phi\right) + \frac{1}{2}\sin\left(2\Phi + \frac{4\pi}{3}\right) + \frac{1}{2}\sin\left(2\Phi - \frac{4\pi}{3}\right) = 0$$

Now we can apply the second identity to the second and third terms:

$$\frac{1}{2}\sin\left(2\Phi\right) + \frac{1}{2}\left[\sin\left(2\Phi\right)\cdot\cos\left(\frac{4\pi}{3}\right) + \cos\left(2\Phi\right)\cdot\sin\left(\frac{4\pi}{3}\right) + \sin\left(2\Phi\right)\cdot\cos\left(\frac{4\pi}{3}\right) - \cos\left(2\Phi\right)\cdot\sin\left(\frac{4\pi}{3}\right)\right] = 0$$

We can combine terms to simplify it:

$$\frac{1}{2}\sin\left(2\Phi\right) + \sin\left(2\Phi\right)\cdot\cos\left(\frac{4\pi}{3}\right) = 0$$

Replacing the cosine with its value gives us the following:

$$\frac{1}{2}\sin\left(2\Phi\right) - \frac{1}{2}\sin\left(2\Phi\right) = 0$$

This, of course, simplifies to this obviously true expression:

$$0 = 0$$

Now we just need to show that the sum of the scaled y components is always positive. The y components of each of the three microphones are as follows:

$$y_1 = \sin\left(\Phi\right)$$

$$y_2 = \sin\left(\Phi + \frac{2\pi}{3}\right)$$

$$y_3 = \sin\left(\Phi - \frac{2\pi}{3}\right)$$

Scaling these by their delays and summing them gives the following:

$$C\cdot\sin\left(\Phi\right)^2 + C\cdot\sin\left(\Phi + \frac{2\pi}{3}\right)^2 + C\cdot\sin\left(\Phi - \frac{2\pi}{3}\right)^2 > 0$$

C is a positive value, so each term is positive or 0. This means we just need to ensure that all three terms can't be 0 at the same time.

Let's set Φ to 0 to make the first term 0:

$$C\cdot\sin\left(\frac{2\pi}{3}\right)^2 + C\cdot\sin\left(-\frac{2\pi}{3}\right)^2 > 0$$

After evaluating the sine function, we get this:

$$\frac{C\cdot 3}{2} > 0$$

This is true, as C is positive.

If we set Φ to $\pm\frac{2\pi}{3}$, we will end up with the same result by symmetry.

Therefore, no matter what angle the sound comes from, the sum of the scaled vectors will point toward it.

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Z